Naval Research Laboratory

Washington, DC 20375-5320

NRL/FR/5583--96-9810

Methods for Generating Synthetic Databases with Specified Statistical Properties

KAREN A. ERNER

Advanced Information Technology Branch
Information Technology Division

March 27, 1996

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave Blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 27, 1996	Interim Report

4. TITLE AND SUBTITLE

Methods for Generating Synthetic Databases with Specified Statistical Properties

5. FUNDING NUMBERS

AO-A613-00
WU-55-3807-00

6. AUTHOR(S)

Karen A. Erner

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/FR/5583--96-9810

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Advanced Research Projects Agency
OOTW/LE
3701 North Fairfax Drive, Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

U.S. Department of Treasury
Office of Systems Investigation
Artificial Intelligence Division, Washington, DC

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

I have developed methods for the generation of a synthetic database, whose records may include fields of different types text strings, characters, and numbers. I have examined techniques that allow us to simulate particular frequency distributions for each field in a data record and to reflect relations between the fields in a data record. In addition, to produce more realistic looking text data. I have also presented an elementary algorithm that generates key entry or typographical errors.

14. SUBJECT TERMS

Database
Synthetic data

15. NUMBER OF PAGES

75

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std 239-18
298-102

CONTENTS

METHODS FOR GENERATING SYNTHETIC DATABASES WITH SPECIFIED STATISTICAL PROPERTIES

INTRODUCTION

In this report, I develop methods for the generation of a synthetic database, whose records may include fields of different types including text strings, characters, and numbers. I also explore schemes that allow us to relate fields within a record or between tables and introduce realistic inconsistencies in the data by simulating key entry errors.

The ability to generate a synthetic database is always useful, whenever the real data are sensitive. Real data may be either classified or may be subject to laws governing the Privacy Act. It is always possible to create a filter that distorts data by mapping the actual values into others. However, in some cases even the relationships between fields or tables may be deemed as sensitive information. Therefore such a distortion may not always be possible.

Also, in the case of distorted real data, we have no control over the relationships between the data fields or tables. If we generate completely random data, we would perceive no true relationships at all.

The capability to assemble such a synthetic database allows us to explore new database models before preparing or disseminating the actual data. Since we know the field and table relations beforehand, we may also properly test database query functions and statistical analysis tools. With the introduction of key entry errors, we may also provide synthetic data that challenge "fuzzy" queries. For example, it lets us test LIKE queries that are traditionally based on the Soundex algorithm. Originated by Margaret K. Odell and Robert C. Russell, the Soundex algorithm matches similarly sounding strings that begin with the same letter [1].

I proceed by looking at the algorithms that produce our data fields in the next section, and I implement them by using the C and C^{++} programming languages. Here I discuss various techiques that create relationships between data fields and simulate key entry errors.

This report has four appendices that contain the complete C and C^{++} source code for the synthetic database generator developed herein. These four appendices cover: random functions, distribution tables, key entry errors, and a simple text string class.

METHODS

The records of any database comprise many types of data; such as numbers, characters, and text. In creating synthetic data, we may use several strategies dependent on the type of data. I implement these strategies by using two techniques based on a random number generator: numeric data

Manuscript approved January 18, 1996.

generation and table look-up methods. Before discussing these methods, lets take a very brief look at the random number generator.

Random Number Generator

All of our synthetic database methods rely on our ability to generate random numbers in the interval from 0 to 1.0. Because we will call this function often, we want to use as few execution steps as possible and still produce a good uniform distribution. The design of random number generators for speed, portability, and goodness of fit is a field unto itself, so I choose not to elaborate on these algorithms in this report. For simplicity, I use the ANSI C rand() function that generates unsigned integers between 0 and the maximum number, RAND_MAX. The Random() function below divides the result of rand() by RAND_MAX to produce double precision float point numbers between 0 and 1.0.

```
double Random(){
    return (double)rand()/(double)RAND_MAX;
};
```

We also want to be able to reseed our random number generator, so I have included a seeding function that uses the ANSI C srand() function. This function takes an unsigned integer as its argument to seed the generator.

```
unsigned Srandom(unsigned myseed){
    srand(myseed);
    return myseed;
};
```

The ANSI C rand() function is certainly not the best random number generator available, however in declaring the Random() and Srandom() functions, I have provided an interface that allows us to substitute other random number and seeding methods into the function definitions without having to change the numeric data generation and table look-up routines.

Numeric Data Generation

To fabricate simple numeric data, I use two basic methods. The first method generates numbers from an average value, while the second method produces numbers and characters from specified ranges.

Numbers from Average Values

We can always select numbers for average values using Gaussian or other distribution techniques. These techniques involve a fair amount of computation and must be used if we wish to realize a particular continuous distribution. For example, we might have a data field where we wish to reflect a mean value of 5. To produce Gaussian or Normal deviates, we might use the following function based on gasdev() from *Numerical Recipes* [2]:

```
double Gaussian(double mean){
    static int iset = 0;
    static double gset;
    double fac, r, v1, v2;

        if (iset == 0) {
        do {
            v1=2.0*Random()-1.0;
            v2=2.0*Random()-1.0;
            r=v1*v1+v2*v2;
        } while (r >= 1.0);
        fac=sqrt(-2.0*log(r)/r);
        gset=v1*fac;
        iset=1;
        return (v2*fac + mean);
      } else {
        iset=0;
        return (gset + mean);
    }
};
```

The Gaussian() function requires several execution steps. It calls the random number generator twice within a while loop and uses both the square root and logarithm functions.

We can also select numbers from average values by using an ad hoc process that we call "randomization." If we are given a mean value μ, our Randomize() function selects a random number from the range

$$\mu - \sqrt{\mu}\ ,\ \mu + \sqrt{\mu}\ .$$

This is a less computationally expensive method and for many situations randomization will suffice. We need only make a single call to the random number generator and use just the square root function. Implemented with the C programming language, our Randomize() function looks like this:

```
double Randomize(double mean){
    double value;
    value = Random() - 0.5;          // generate a random number between -0.5 and 0.5
    value *= (2.0 * sqrt(mean));     // multiply the random number by twice the square
                                     // of the mean
    value += mean;                   // add the mean
    return value;
};
```

Figure 1 compares the values produced by the Gaussian() and Randomize() functions.

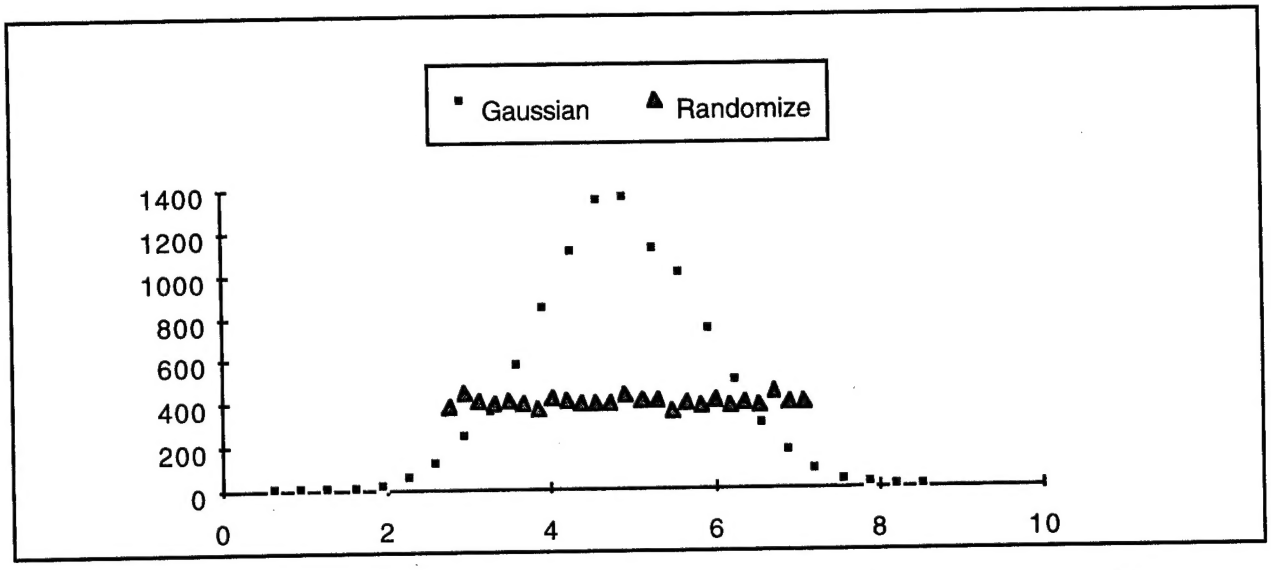

Fig. 1 — Histogram of deviates using the Gaussian() and Randomize() functions with a mean of 5. There were 10,000 trials and 25 histogram bins.

The Randomize() function merely produces a uniform distribution centered about the mean with a width equal to twice the square root of the mean, so it does not reproduce the shape of a Gaussian distribution, including its tails. It does, however, have the advantage of speed over the Gaussian() function. Figure 2 shows the timing results using the two functions on an IBM 486DX 66 MHz computer. The Randomize() function executes roughly twice as fast as the Gaussian() function.

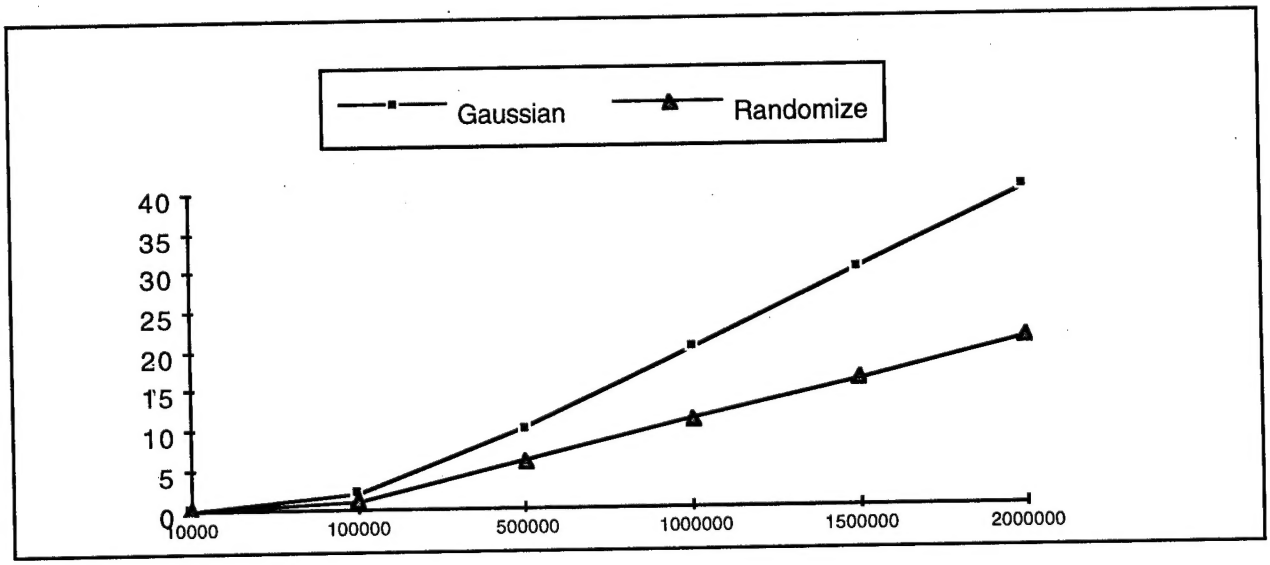

Fig. 2 — Timing in seconds vs trials for the Gaussian() and Random() functions using 10,000 to 2,000,000 iterations on an IBM 486DX 66 MHz computer.

If we need to create a large number of database records with numeric fields representing average values and if the shape of the distribution is not an issue, the Randomize() function cuts down the runtime considerably.

We may also consider character data to be numeric. On most machines, alphabetic characters map to their ASCII values, which are integers in the ranges of 65-90 for upper case and 97-122 for lower case. We might use randomization to generate alphabetic characters, however at the extrema of the character range, our ASCII values may produce nonalphabetic results. Usually we do not want this feature, although for some applications it may be desired. For character fields, we prefer using either range or table look-up methods.

Numbers and Characters from Specified Ranges

To fill in numeric and character data fields, we can select values from a uniform distribution with defined ranges. As an example, we might have a data field representing the income of an individual and we would like to generate incomes in the range of $10,000 to $100,000.

We want to build a function, Rrange(), that selects a random number from a range, given the extrema as arguments.

```
minimum = 1000;
maximum = 100000;
number = Rrange(minimum, maximum).
```

The algorithm for such a function proceeds as follows:

1. *Determine which argument is the minimum and which is the maximum.*

2. *Find the difference between the minimum and the maximum.*

   difference = maximum - minimum

3. *Generate a random number between 0 and 1.0.*

4. *Multiply the random number by the difference.*

   *random number = random number * difference*

5. *Add the minimum value to the random number*

   random number = random number + minimum

We may readily implement this algorithm by using the C or C++ programming languages.

```
double Rrange(double d1, double d2){
    double min;
    double delta;
    double rn;

    // Determine which argument is the minimum
    min = (d1 < d2) ? d1 : d2;

    // Find the absolute difference between the arguments
    delta = fabs(d1 - d2);
```

```
    // Draw a random number between 0 and 1 and multiply by the absolute difference
    rn = Random();
    rn *= delta;

    // Add this random number to the minimum value in the range and return the result
    rn += min;
    return rn;
};
```

In the case of characters, the derivation of similar functions follows immediately based on the ASCII character set. We might have a function for lower case with an integer range of 97-122, and another for upper case with an integer range of 65-90.

```
character = Rrange('a','z');
character = Rrange('A','Z');
```

When generating characters, we must apply this method with caution as it may rely on the ASCII character set, while some machines employ the EBCDIC character set instead. Using C++, it is safest to overload the Rrange() function for characters using the appropriate casts [3].

```
char Rrange(char minimum, char maximum){
    return (char)(Rrange((double) minimum, (double) maximum);
};
```

Table Look-Up Functions

We can generate any data field by looking up our field values from a table or list. For example, if we wish to fill a field for a person's occupation, we might randomly choose from a list like:

MANAGER
SALES
SECRETARY
CLERK
LAWYER
DOCTOR
UNEMPLOYED
COMPUTER PROGRAMMER
TELLER
MECHANIC.

As another example for numeric data types, we might wish to break up our earlier [$10,000, $100,000] individual income range into the following amounts:

10000
15000
30000
50000
100000.

However, if we use simple tables like these we will only produce a uniform distribution from our occupation and income tables. For our first example, there will be just as many managers, as sales personnel, as secretaries and so forth. Such a scheme would not properly reflect the distribution of occupations across a given population. Likewise, from our second example in generating incomes, we will have the same number of people with earnings of $10,000, $15,000, $20,000 and so on. Further, there would be no variation in income amounts unless we randomize the results of our table look ups. We would prefer to express our tables in terms of frequencies, (Table 1).

Table 1 — Occupations and Frequencies

OCCUPATION	FREQUENCY
MANAGER	5
SALES	10
SECRETARY	7
CLERK	10
LAWYER	2
DOCTOR	1
UNEMPLOYED	1
COMPUTER PROGRAMMER	2
TELLER	3
MECHANIC	4

From Table 1 we see that in a total population of 45 individuals, there are five managers, 10 sales personnel, seven secretaries, 10 clerks and so forth.

Similarly, for our income table we might use the frequencies (Table 2) and then randomize our results. Here 11 people will have average incomes of $10,000, while 15 people will have average incomes of $15,000 and so forth. Unfortunately, with this approach, since our Randomize() function has a range defined by the square root of the mean value, then 11 people will have incomes anywhere in the range [$9,900, $10,100] and 15 people will have earnings in the range [$14,878, $15,123]. No one could earn an annual salary of $12,537. Obviously, this method does not produce a continuous spectrum of incomes.

Table 2 — Average Incomes with Corresponding Frequencies

AVERAGE INCOME	FREQUENCY
10000	11
15000	15
30000	20
50000	7
100000	2

Instead, we can take advantage of our random range function, Rrange(), and treat the values in our list as salary intervals. Thus, we can have 11 people with earnings from 0 to $10,000 and 15 people earning from $10,000 to $15,000.

To meet the needs demonstrated by our examples above, I define two types of tables: a look-up table, returning exact values from a list, and an interval look-up table, returning a random value from some range in the list. I will describe these tables and their C++ language implementations next.

Look-Up Tables

Using a simple look-up table, we may select items from a list with corresponding frequencies through a random key or look-up value. Our table items may consist of any object or data type. For instance, we might choose to assemble number, character or character string lists. In general, we wish to access a table of N data items with N corresponding frequencies.

<item 1>	*<frequency 1>*
<item 2>	*<frequency 2>*
<item 3>	*<frequency 3>*
.	.
.	.
.	.
<item n>	*<frequency n>*

From the frequencies f, we may calculate the distribution of our list where the distribution value d for the mth item in the list is given by

$$d_m = \sum_{i=0}^{m-1} f_i .$$

We further normalize our distribution values by using the sum of the N frequencies F.

$$F = \sum_{i=0}^{N} f_i$$

$$norm(d_m) = \frac{d_m}{F} .$$

Now, if we generate a random number rn, from zero to one, we can look up values from our list. If $rn > norm(d_{m-1})$ and $rn \le norm(d_m)$, then select item m from the list.

We can implement a look-up table in C++ by first defining a structure for each item in the table that contains the statistical frequency and distribution values. Let us call this the Stats structure:

```
struct Stats {
    double frequency;
    double distribution;
    Stats(double f = 0.0, double d = 0.0) : frequency(f), distribution(d) {};      // constructor
    ~Stats(){};                                                                    // destructor
    friend ostream& operator<<(ostream& os, Stats& s);
    friend istream& operator>>(istream& is, Stats& s);
};
```

The Stats structure contains two data members, one for the frequency and one for the distribution. Its constructor has default values of 0.0 for both the frequency and the distribution. I have also endowed the Stats structure with overloaded ostream and istream operators.

```
ostream& operator<<(ostream& os, Stats& s){
    os << s.frequency << "\t" << s.distribution;
    return os;
};

istream& operator>>(istream& is, Stats& s){
    is >> s.frequency;
    return is;
};
```

Notice that while the output stream operator prints both the frequency and distribution values, the input stream operator only reads the frequency. We compute the distribution value when we load a table at runtime. This simplifies the maintenance on our data lists, enabling us to add or delete items to the list along with their frequencies without having to adjust the overall distribution values. For instance, we might add other occupations and frequencies to our occupation list. I chose to implement Stats as a structure rather than a class for this very reason.

Using our Stats structure along with C++ templates, we can construct a look-up table class for any item type or object.

```
template <class T>
class Table {
public:
Table(int e = 0);                          // constructor
Table(char* filename);                     // file constructor
~Table();                                  // destructor
int getElements() { return elements; }     // return the number of elements
Table& normalize();                        // normalize table statistics
const T& lookUp(double k) const;           // look up table item
const T& getItem(int i) const;             // return the ith item in the list
friend ostream& operator<<(ostream& os, Table& t);
friend istream& operator>>(istream& is, Table& t);
protected:
    int elements;                          // number of table elements or items
    T* item;                               // pointer to table elements
    Stats* stats;                          // pointer to corresponding statistics
Table& findDistribution();                 // calculate distribution
};
```

The Table class has three private data members: elements, item, and stats. Table::elements is the integer number of table elements or items, while Table::item is a pointer to the list of table items. The Table::stats member is a Stats pointer to statistics corresponding to the list of items referred to by the Table::item pointer.

I've given the Table class two constructors. The first table constructor serves as the default constructor and takes the number of elements as an argument.

```
template <class T>
Table<T>::Table(int e){
    elements = e;
    if (elements < 0){
        cerr << "table elements < 0" << endl;
        exit(1);
    }
    else if (elements > 0){
        item = new T[elements];
        stats = new Stats[elements];
    }
    else {
        item = 0;
        stats = 0;
    }
};
```

This constructor tests to make sure that the number of elements specified by the argument is not less than zero. If the number of elements is less than zero, it issues an error message and exits the program. Otherwise it allocates the item array and the corresponding stats array to a size equal to the number of elements, or sets Table::item and Table::stats to NULL when the number of elements is zero.

Since we typically load our Tables with file data, I have included a file constructor that uses the name of the data file as its argument.

```
template <class T>
Table<T>::Table(char* filename){

    ifstream fin;

    fin.open(filename);

    if (!fin){
     cerr << "Table constructor:  cannot open " << filename << endl;
     exit(1);

    }

    fin >> elements;

    if (elements <= 0){
     cerr << "Table constructor:  elements <= 0)" << endl;
     exit(1);

    }

    item = new T[elements];
    stats = new Stats[elements];

    for (int i = 0; i < elements; i++)
        fin >> item[i] >> stats[i];

    normalize();

    fin.close();
};
```

The file constructor attempts to open the data file specified by its argument. If it cannot open the file, it issues an error message and exits the program. Next, it reads in the number of table elements from the file. The constructor considers a Table::elements value that is less than or equal to zero an error. If the number of elements is nonzero, space is allocated for the Table::item and the Table::stats arrays. Finally, it reads the data into the table, normalizes the table, and then closes the input file.

The Table destructor deallocates any memory taken from the free store.

```
template <class T>
Table<T>::~Table(){
    if (elements){
        delete[] item;
        delete[] stats;
        item = 0;
        stats = 0;
        elements = 0;
    }
};
```

Our Table class contains a private method, findDistribution(), to calculate the table distributions. This function maintains a running sum of the table frequencies and assigns the current distribution values accordingly.

```
template <class T>
Table<T>&
Table<T>::findDistribution(){
    int i;
    double sum = 0.0;

    for (i = 0; i < elements; i++){
        sum += stats[i].frequency;
        stats[i].distribution = sum;
    }

    return *this;
};
```

The public Table normalize() function calls findDistribution() before normalizing the table's statistics. It uses the maximum distribution value as the normalization factor and tests for a nonzero value before normalization of the table statistics.

```
template <class T>
Table<T>&
Table<T>::normalize(){
    int i;
    double nfactor;

    findDistribution();

    nfactor = stats[elements-1].distribution;
```

```
    if (nfactor == 0){
        cerr << "table normalize:  divide by zero" << endl;
        exit(1);
    }

    else {
        for (i = 0; i < elements; i++){
            stats[i].frequency /= nfactor;
            stats[i].distribution /= nfactor;
        }
    }
    return *this;
};
```

Using a binary search algorithm, the Table lookUp() function returns a constant reference to an item from a list given a key of type double from the range [0.0, 1.0]. This function is constant. It cannot alter the Table's data.

```
template <class T>
const T&
Table<T>::lookUp(double k) const {
    int minimumIndex = 0;
    int maximumIndex = elements - 1;
    int middleIndex = 0;

    // test key range
    if (k < 0.0 || k > 1.0){
        cerr << "Table::lookUp() - key out of range" << endl;
        exit(1);
    }

    // if key is 0.0, return item at minimum index
    if (k == 0.0)
     return (item[0]);

    // if key is 1.0, return item at maximum index
    if (k == 1.0)
     return (item[maximumIndex]);

    while (maximumIndex >= minimumIndex){
        middleIndex = (maximumIndex + minimumIndex)/2;
        if (k > stats[middleIndex].distribution)
         minimumIndex = middleIndex+1;
        else if (k < stats[middleIndex].distribution)
         maximumIndex = middleIndex-1;
        else {
         return (item[middleIndex]);
        }
    }
    return (item[minimumIndex]);
};
```

It's probably best to explain how the lookUp() function operates by way of an example. Suppose we have a Table with six elements, whose item array contains the objects A, B, C, D, E and F. These items correspond to a distribution array whose values are 0.2, 0.4, 0.5, 0.6, 0.8 and 1.0, and we wish to find an item in the table using a key equal to 0.82.

The lookUp() function keeps track of three array indices: the minimum index (minimumIndex), the middleIndex (middleIndex) and the maximum index (maximumIndex.) It initially sets the minimum and middle indices to zero and the maximum index to the number of table elements minus one. (In the C and C++ programming languages arrays are indexed from zero and not one.) To start, minimumIndex = middleIndex = 0 and maximumIndex = 5. Schematically, our problem looks like this:

0	1	2	3	4	5
A	B	C	D	E	F
0.2	0.4	0.5	0.6	0.8	1.0

↑ min, mid ↑ max

Initially, the lookUp() function tests to see if the key is in the range from 0 to 1.0. If not, it issues an error message and terminates the program. Next, it checks to see if the key is equal to either 0.0 or 1.0. If so, the function returns the table item located at the minimum or maximum array index respectively. Since our key is 0.82, this is not the case, so we proceed into a while loop.

To find the correct table item, the minimum and maximum index values must converge, so that the while loop uses the condition:

```
while (maximumIndex >= minimumIndex) { ... };
```

Within the while loop, the first step is to calculate the middle index via integer arithmetic.

```
middleIndex = (maximumIndex + minimumIndex) / 2;
```

For our first trip through the loop:

middleIndex = (5 + 0)/ 2 = 2.

0	1	2	3	4	5
A	B	C	D	E	F
0.2	0.4	0.5	0.6	0.8	1.0

↑ min ↑ mid ↑ max

Next the function compares the argument key to the distribution value at the middle index. At this point the middle index distribution value is 0.5. Since our key 0.82 is greater than 0.5, the lookUp() function recalculates the minimum index as follows:

minimumIndex = middleIndex + 1,
minimumIndex = 2 +1 = 3.

0	1	2	3	4	5
A	B	C	D	E	F
0.2	0.4	0.5	0.6	0.8	1.0
		↑ mid	↑ min		↑ max

We will go through the while loop again since its condition is still met. The maximum index 5 is greater than or equal to the minimum index 3. So we compute the value of the middle index once again:

middleIndex = (maximumIndex + minimumIndex)/2 = (5 + 3) = 4.

0	1	2	3	4	5
A	B	C	D	E	F
0.2	0.4	0.5	0.6	0.8	1.0
		↑ min		↑ mid	↑ max

Again our look-up key 0.82 is greater than the distribution value at the middle index 0.8 and we recalculate the minimum index.

minimumIndex = middleIndex + 1 = 4 + 1 = 5.

0	1	2	3	4	5
A	B	C	D	E	F
0.2	0.4	0.5	0.6	0.8	1.0
				↑ mid	↑ max min

At this point, the minimumIndex and the maximumIndex are both equal to 5, therefore the while loop condition is false and the lookUp() function returns the item whose index is equal to the minimum index. Thus for this example, with a key of 0.82, our lookUp() function returns item F.

Admittedly, our "divide-and-conquer" binary search lookUp() function may look like a bit of overkill for such a small table. At this scale, it might be both easier and more efficient to use a sequential search routine. However I have assumed that the database developer requires the ability to generate a great deal of data and therefore the input frequency tables are likely to be large, justifying the extra complexity.

The choice of search algorithms depends on the size of the input tables. A sequential search yields a worst-case execution time proportional to N, the number of elements in the table, whereas a binary search uses a runtime no longer than log(N) [4]. The synthetic database generator should use only sequential searches when the frequency tables are very small.

To handle input and output, the Table class overloads the ostream and istream operators.

```
template <class T>
ostream& operator<<(ostream& os, Table<T>& t){
    if (t.elements){
      for (int i = 0; i < t.elements; i++)
        os << t.item[i] << "\t" << t.stats[i] << endl;
    }
    return os;
};
```

The Table ostream operator << writes its table elements along with their respective statistics to the output stream.

```
template <class T>
istream& operator>>(istream& is, Table<T>& t){
    int e;
    is >> e;

    if (e <= 0){
      cerr << "table elements <= 0" << endl;
      exit(1);
    }

    if (e != t.elements && t.elements > 0){
        delete[] t.item;
        delete[] t.stats;
    }

    t.elements = e;
    t.item = new T[t.elements];
    t.stats = new Stats[t.elements];

    for (int i = 0; i < t.elements; i++)
        is >> t.item[i] >> t.stats[i];

    t.normalize();

    return is;
};
```

The istream operator >> first reads the number of table elements. If its elements are currently nonzero and not equal to the input elements, it deallocates its item and stats arrays and then reallocates arrays of the appropriate size. After reading in the table data, it calls the normalize() function.

Here is a sample program that reads an input file containing our list of occupations and their frequencies. It seeds the random number generator and produces 500 random keys to look up occupations from the table and prints the resulting occupations to an output file.

```
int main(){

    int i;
    int trials = 500;                    // number of trials

    double key;                          // table look-up key

    unsigned myseed = 2312;              // seed for random number generator

        Srandom(myseed);                 // seed the generator

    fstream fout;                        // output file stream

    // Create a table for occupations using data from the jobs.txt file
    Table<String> JobTable("jobs.txt");

    // open the output file
        fout.open("data.txt", ios::out);
    if (!fout){
        cerr << "cannot open data.txt" << endl;
        exit(1);
    };

    // generate random keys between 0 and 1, look up table values and print the
    // results to the output file.
        for (i = 0; i < trials; i++){
        key = Random();
        fout << JobTable.lookUp(key) << endl;
    }

    // close the output file
        fout.close();

    return 0;

};
```

In the program above, we created and loaded file data in one step by using the Table file constructor. Otherwise, we would have had to use the default constructor, open a file, read the data and close the file.

Figure 3 shows a histogram containing our occupation results, normalized and rescaled to the theoretical values from our input table.

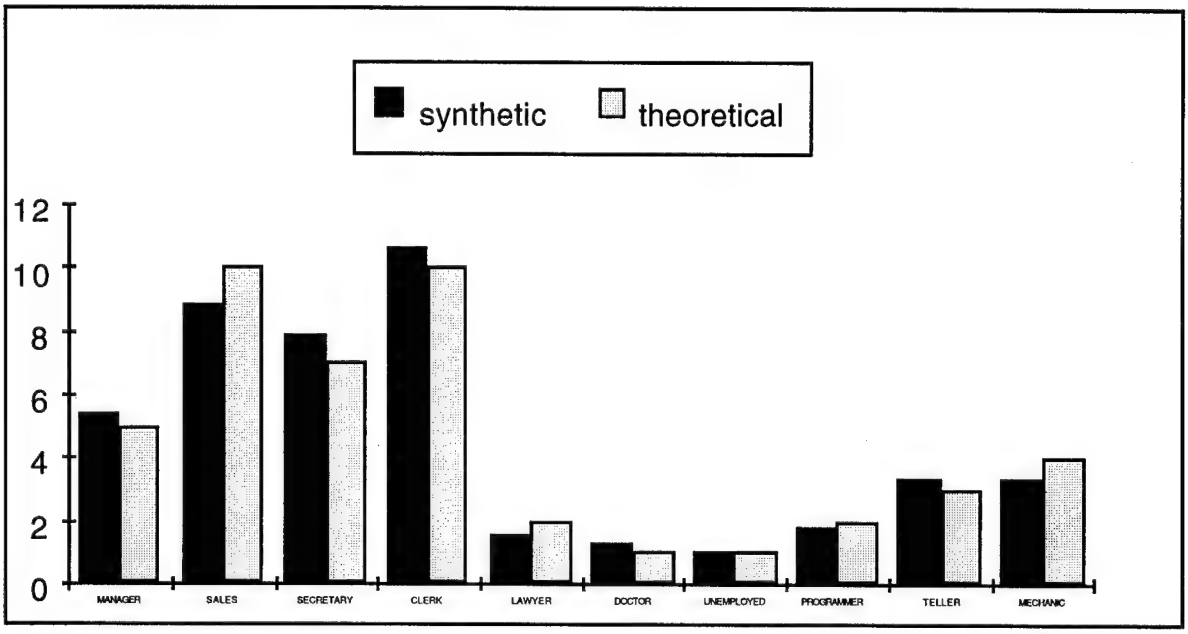

Fig. 3 — Histogram of generated occupations vs theoretical occupations using the table look-up method and 500 trials. The synthetic data was normalized and then scaled to the theoretical input data.

Though not an exact match, the distribution of our synthetic data agrees rather well with our theoretical distribution over 500 trials.

Next, we look at interval look-up tables that allow us to generate variable and continuous numeric data.

Interval Look-Up Tables

If our field items in a table consist of numeric types, we probably do not wish to return a specific number from the list. For numerical list look ups, we combine the table look-up method with the Range() function (Table 3). From our earlier example (Table 2), we may wish to break up our [$10,000, $100,000] income range into the following frequency list:

Table 3 — Income Range Values with Frequencies

INCOME RANGE	FREQUENCY
10000	11
15000	15
30000	20
50000	7
100000	2

If we generate a random number *rn* and then look up results in an income of $30,000 from the table, we may use our Rrange() function to return a random income in the range from $15,000 to $30,0000.

If $rn > norm(d_{m-1})$ and $rn \leq norm(d_m)$, then the return value exists in the range

$$(item_{m-1}, item_m].$$

We can quickly implement an interval look-up table using the C++ programming language by deriving this class from the Table class described in the last section.

```
template <class T>
class IntervalTable : public Table<T> {
public:
IntervalTable(int e = 0) : Table<T>(e) {};          // constructor
IntervalTable(char* filename) : Table<T>(filename) {};   // file constructor
T lookUpInterval(double k) const;                   // look up random table value
};
```

The only difference between the two classes is the addition of the lookUpInterval function. This function performs a binary search like the Table::lookUp() function described earlier, but calls Rrange() to return a random item based on the intervals in the frequency table.

```
template <class T>
T
IntervalTable<T>::lookUpInterval(double k) const {
    int min, max;
    int minimumIndex = 0;
    int maximumIndex = elements - 1;
    int middleIndex = 0;
    T zero = 0;

    if (key < 0.0 || key > 1.0){
     cerr << "IntervalTable::lookUpInterval() - key out of range" << endl;
     exit(1);
    }

        // If key is 0.0, return zero
    if (k == 0.0)
     return zero;

    // If key is 1.0, return a random value between the maximum
    // index and the next lower index.
    if (k == 1.0){
     max = maximumIndex;
     min = max - 1;
     if (min < 0)
         return (Rrange(zero, item[max]));
     return (Rrange(item[min], item[max]));
    }
```

```
    while (maximumIndex >= minimumIndex){
        middleIndex = (maximumIndex + minimumIndex)/2;

        if (k > stats[middleIndex].distribution)
          minimumIndex = middleIndex+1;
        else if (k < stats[middleIndex].distribution)
          maximumIndex = middleIndex-1;
        else {

          max = middleIndex;
          min = max - 1;
          if (min < 0)
              return (Rrange(zero, item[max]));
          return (Rrange(item[min], item[max]));
        }
    }
    max = minimumIndex;
    min = max - 1;
    if (min < 0)
      return (Rrange(zero,item[max]));
    return (Rrange(item[min], item[max]));
};
```

You must instantiate an IntervalTable object using a numeric data type or class for which you have defined a random range, Rrange() function. Since we don't have an Rrange() function for the String class, though the declaration of a String IntervalTable would be legitimate, an attempt to call the lookUpInterval() function will result in a compiler error.

```
IntervalTable<String> JobTable("jobs.txt");          // okay

JobTable.lookUpInterval(mykey);                      // error! Rrange(String, String)
                                                     // not defined!
```

Here is a sample program that uses our salary frequency table. It loads the salary data, seeds the random number generator and creates 100,000 values by using the table and prints them to standard output.

```
int main() {
    unsigned myseed = 32;              // seed for random number generator

    int i;
    int trials = 100000;               // number of trials

    double mykey;                      // table look-up key

    // Salary IntervalTable is created using data from the salary.txt file.
    IntervalTable<long> salaries("salary.txt");

    // Seed the random number generator
    Srandom(myseed);
```

```
    for (i = 0; i < trials; i++){
        mykey = Random();              // Generate a random key between and 1

        // use key to look up salary from the salary table and print to standard output
        cout << salaries.lookUpInterval(mykey) << endl;
    }

    return 0;
};
```

Figure 4 charts a histogram of our synthetically generated salary results and compares it to the theoretical data supplied by the salary frequency table. The IntervalTable look-up function performed very well over 100,000 trials.

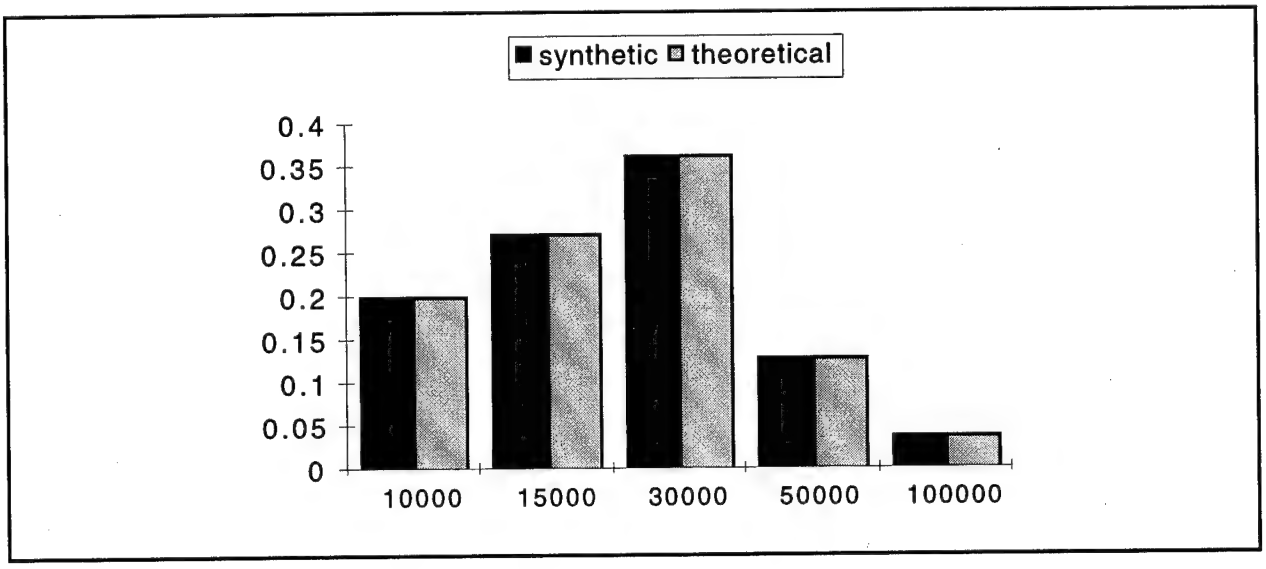

Fig. 4 — Normalized histogram of synthetically generated earnings vs theoretical salaries using the IntervalTable LookUpInterval() function over 100,000 trials.

If we use all of the above methods, we can certainly generate a database wherein each field within each record is created randomly. However, in doing so, the fields of our synthetic database will remain independent and will only reflect their native distributions, and bear no relationship to one another. If our intention is to test a relational database model or to evaluate database analysis tools, our purely random data may certainly serve as a benchmark for read/write and for basic query utilities, but for little else. We must be able to establish dependencies between our synthetic data fields.

Simulating Field Dependencies

We wish to develop a method where we can reflect the dependency of one data field upon another field or a group of data fields upon another field. For instance,

FIELD A $\Rightarrow$ *FIELD B*

FIELD A + FIELD B $\Rightarrow$ *FIELD C.*

We also want to derive methods where the dependency of one field is sometimes dependent on another field or on a group of fields and other times not.

*FIELD A * PROBABILITY A* $\Rightarrow$ *FIELD B*

*(FIELD A * PROBABILITY A) + (FIELD B * PROBABILITY B)* $\Rightarrow$ *FIELD C.*

To create these dependencies, we use a simple hash function. Hash functions allow us to replace a random number as our look up key, with a number based on a character string's hash value. Our hash function below uses the modulus operator and Horner's method to generate an unsigned integer from a character string [5]:

```
unsigned Hash(char* mystring){
    unsigned j;
    unsigned keysize;
    unsigned h;

    keysize = (unsigned)strlen(mystring);      // find the length of the string
    h = (unsigned)mystring[0];                 // get the first character in the string

    for (j = 1; j < keysize; j++)
        h = (h*37) + (unsigned)mystring[j];

    h %= RAND_MAX;

    return h;
};
```

Next, we create a hash function, Dhash() that uses the Hash() function above to generate a table look-up key in the range from 0 to 1.0.

```
double Dhash(char* mystring){
    unsigned h;
    double r;

    h = Hash(mystring);

    r = (double)h/(double)RAND_MAX;

    return r;
};
```

Our Dhash() function takes a string as an argument, calls the Hash() function and then returns a double float value, normalized using the maximum random number, RAND_MAX.

We may use the Dhash() function to represent the dependency between one field and another by generating a key that uses one field and then by using this key as the look-up value for the dependent field. For instance, if we have two fields: occupation and hobby, and we want to establish a relationship between occupation and hobby, we might use the following code segment:

```
double hobby_key;
String occupation;
String hobby;

// Generate a look-up key for hobby using occupation
hobby_key = Dhash(occupation);

Table<String> HobbyTable;

// Look up hobby in table using the hobby key
hobby = HobbyTable.LookUp(hobby_key);
```

One occupation may be CLERK and another may be MECHANIC. The text string "CLERK" hashes to 0.634816, while the text string "MECHANIC" hashes to 0.366253. Using these different numeric values, we may look up hobbies in a hobby table (Table 4). For example, where after calculation and normalization of the frequency and the distribution, we have (Table 5). Since "CLERK" hashed to 0.634816 our hobby table look-up would find "CRAFTS", while the text string "MECHANIC" that hashes to 0.366253, would produce the hobby "JOGGING". This reproduces the relation

$$FIELD\ A \Rightarrow FIELD\ C,$$

that is typically a many-to-one mapping as you can see in Table 6. Both MANAGER and COMPUTER PROGRAMMER result in MOVIES as a hobby, and DOCTORS, TELLERS and MECHANICS participate in JOGGING. Furthermore, we find that four of our 10 hobbies: BIKING, SKIING, COOKING and MUSIC, will never appear in our data, if hobbies are solely dependent on occupation at all times.

Table 4 — Hobbies with Associated Frequencies

HOBBY	FREQUENCY
WRITING	10
BIKING	8
JOGGING	7
BUNGEE JUMPING	2
SKIING	4
STAMP COLLECTING	3
CRAFTS	5
COOKING	7
MUSIC	8
MOVIES	6

Table 5 — Hobbies with Normmalized Frequency and Distribution Values

HOBBY	FREQUENCY	DISTRIBUTION
WRITING	0.166667	0.166667
BIKING	0.133333	0.3
JOGGING	0.116667	0.416667
BUNGEE JUMPING	0.033333	0.45
SKIING	0.066667	0.516667
STAMP COLLECTING	0.05	0.566667
CRAFTS	0.083333	0.65
COOKING	0.116667	0.766667
MUSIC	0.133333	0.9
MOVIES	0.1	1.0

Table 6 — The Results of Hashing on Occupation to Produce a Hash Key, Looking up Hobbies from the Hobby Frequency Table

OCCUPATION	HASH KEY	HOBBY
MANAGER	0.900845	MOVIES
SALES	0.0540483	WRITING
SECRETARY	0.432661	BUNGEE JUMPING
CLERK	0.634816	CRAFTS
LAWYER	0.906888	MOVIES
DOCTOR	0.350932	JOGGING
UNEMPLOYED	0.52028	STAMP COLLECTING
COMPUTER PROGRAMMER	0.91345	MOVIES
TELLER	0.339427	JOGGING
MECHANIC	0.366253	JOGGING

To show a relationship between multiple fields, as in the relation

FIELD A + FIELD B $\Rightarrow$ *FIELD C,*

we need only concatenate the two input strings A and B and hash this composite string to a value in order to look up C. More generally, we may hash any number of strings.

FIELD A + FIELD B + FIELD C + ... $\Rightarrow$ *FIELD N.*

by concatentating them to a single string. We can implement this using ANSI C variable arguments in our Dhash() function. The function below takes the number of strings as the first argument and the character strings as its subsequent arguments.

```
double Dhash(int strc, ...){
    int i;
    int strings;
    va_list ap;
    char** array;
    char catbuffer[buffersize];
```

```
    // set the number of strings
    strings = strc;
    if (strings <= 0)
        return 0;

    // allocate the string array
    array = new char*[strings];

    // start the variable argument list from strc
    va_start(ap,strc);

    // read the argument strings into the string array
    for (i = 0; i < strings; i++){
        array[i] = va_arg(ap,char*);
    }

    // concatenate the strings into the buffer
    strcpy(catbuffer,array[0]);
    for (i = 1; i < strings; i++)
        strcat(catbuffer,array[i]);

    // end the variable argument list
    va_end(ap);

    // delete the string array
    delete[] array;

    return Dhash(catbuffer);
};
```

With this function we can hash more than one field to produce our look-up keys. For example, to emulate a relationship between occupation, hobby and automobile, we might write the following program:

```
String automobile;
String occupation;
String hobby;

// Generate a look-up key for automobile using occupation and hobby
automobile_key = Dhash(2, occupation, hobby);

Table<long> AutomobileTable;

// Look up income in table using the income key
automobile = AutomobileTable.LookUp(income_key);
```

With this technique, the same pair of occupations and hobbies will produce the same make of automobile. As an example, if we hash LAWYER and BIKING, we will always generate 0.68392 as a look-up key, so we will always look up the same automobile from the table.

To solve this problem, we can introduce a random component into the hash function. If we randomize the function, by generating a random number some of the time and a hash value the rest of the time, we can produce a distribution for a given field value that in general reflects its frequency plus the relationships of dependent fields simultaneously. We call this the weighted hash function, Whash():

```
double Whash(char* mystring, double w){
    unsigned h;
    double r, rn;

    if (w <= 0.0 || w > 1.0){
        cerr << "weight must be in range (0,1]" << endl;
        exit(1);
    }

    // generate a random number
    rn = Random();

    // if random number is less than equal to the weight, then hash the string
    if (rn <= w)
        r = Dhash(mystring);
    // if not, return a random number
    else
        r = Random();

    return r;
};
```

The Whash() function takes a character string and a probability or weight as arguments. It first draws a random number. If this number is less than or equal to the probability or weight, the function returns the hash value of the argument character string. Otherwise, Whash() draws a new random number and returns it. This function represents the field relation:

*FIELD A * PROBABILITY A* $\Rightarrow$ *FIELD B.*

As an example, let us once again hash on the occupation field to generate the hobby field. However, this time occupation determines the hobby only 30% of the time. So, we would use this function call to produce our look-up key for hobby:

```
hobby_key = Whash(occupation, 0.3);
```

Figure 5 summarizes the hobbies produced for CLERK with this Whash function.

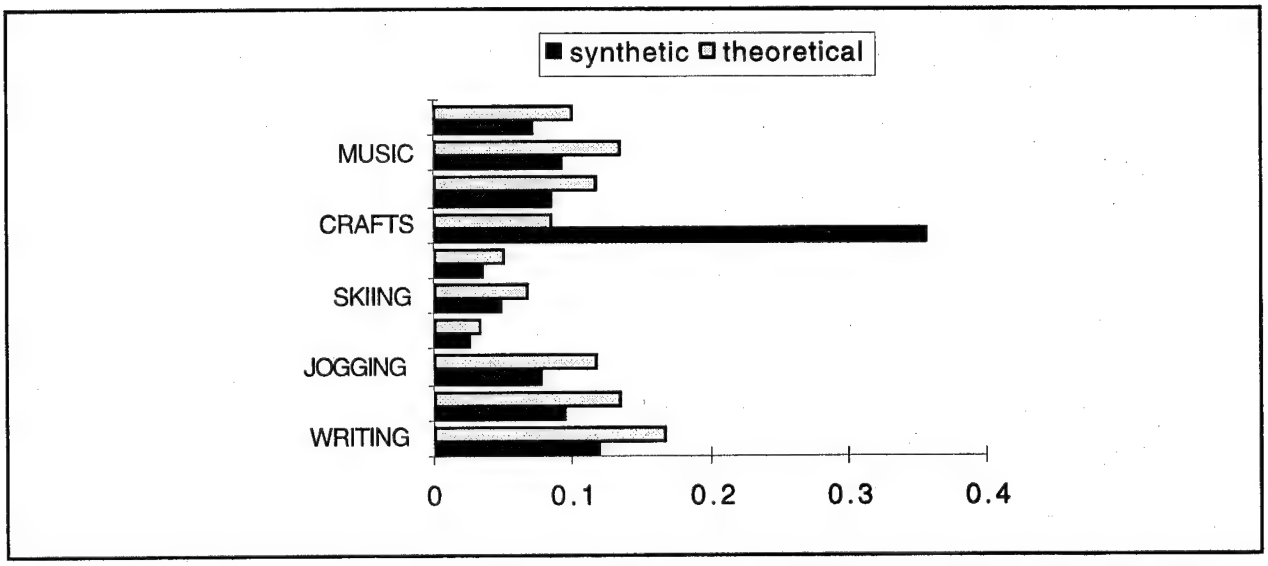

Fig. 5 — Hobbys produced using Whash("CLERK", 0.3) compared with the theoretical input hobby frequencies.

In our earlier example using Dhash(), all clerks had CRAFTS as a hobby. With our new Whash() function, a significant number of clerks have CRAFTS for a hobby as in Fig. 5 above. The distribution of other hobbies reflects that of the input table (Table 5).

If we have a dependency based upon more than one input field

*(FIELD A * PROBABLITY A) + (FIELD B * PROBABILITY B) +...* $\Rightarrow$ *FIELD N,*

we can generalize the Whash function by using ANSI C variable arguments as we did previously with our Dhash() function. In this case, we would define a function of the form:

```
double Whash(int strc, ...){
    int i;
    int strings;
    va_list ap;
    char** array;
    char catbuffer[buffersize];
    double rn;
    double* weight;

    // set the number of strings
    strings = strc;
    if (strings <= 0)
        return 0;

    // allocate the string and weight arrays
    array = new char*[strings];
    weight = new double[strings];

    // start the variable argument list from strc
    va_start(ap, strc);
```

```
    // read the strings and weights into their respective arrays
    for (i = 0; i < strings; i++){
        array[i] = va_arg(ap,char*);
        weight[i] = va_arg(ap,double);
    }

    // end the variable argument list
    va_end(ap);

    // Test the weights.  They must be in the range (0,1].
    for (i = 0; i < strings; i++){
        if (weight[i] <= 0.0 || weight[i] > 1.0){
          cerr << "weight " << i << " must be in range (0,1]" << endl;
          exit(1);
        }
    }

    // set the start of the catbuffer with the NULL character, '\0'
    catbuffer[0] = '\0';

    // for each of the strings and weights
    for (i = 0; i < strings; i++){
        // choose a random number
        rn = Random();

        // if random number is less than or equal to the weight, add
        // the string to the catbuffer.
        if (rn <= weight[i])
            strcat(catbuffer, array[i]);
    }

    // delete the string and weight arrays
    delete[] array;
    delete[] weight;

    // if there are no strings in the buffer, then return a random number
    if (catbuffer[0] == '\0')
        return Random();

    return Dhash(catbuffer);
};
```

The Whash() hash function uses the number of strings plus those strings and their associated weights or probabilities as arguments.

```
double Whash(int strc, char* string1, double weight1, ...);
```

It reads the first argument, the string count (strc), and tests to make sure that its value is greater than zero. If so, it uses this value to allocate two arrays; one to hold the argument strings, and the second to contain their corresponding probabilities. Then, for each string it draws a random number. If the

random number falls below the weight value, then the string is concatentated to a character buffer. If Whash() did not add any of the strings to the character buffer, then it draws a random number between 0 and 1.0 and returns this value. Otherwise, the function returns the hash value of the string buffer.

As an example, lets once again find hobbies, but this time we assume that this field depends on the occupation and the average income with probabilities of 0.3 and 0.5 respectively. In the sample program below, we load our salary data into a standard look-up table instead of an interval look-up table, since we just want to express our field relationship as an average income level. We convert the income values to character strings in order to use them in our hash function.

```
int main() {
    unsigned myseed = 32;              // seed for random number generator

    long i;
    long trials = 100000;              // number of trials
    long myincome;                     // income from table

    double key;                        // look up key

    char jobbuffer[SIZE];              // character buffer for occupation
    char incomebuffer[SIZE];           // character buffer for income

    String jobstring;                  // job string
    String hobbystring;                // hobby string

    // Seed the random number generator
    Srandom(myseed);

    // Create income table and load from the salary.txt file.
    Table<long> income("salary.txt");

    // Create occupation table and load from the job.txt file
    Table<String> job("job.txt");

    // Create hobby table and load from the hobby.txt file
    Table<String> hobby("hobby.txt");

    for (i = 0; i < trials; i++){
        // Find job
        key = Random();                          // Draw random number
        jobstring = job.lookUp(key);             // Look up job in table
        strcpy(jobbuffer,jobstring.getString()); // Write job to buffer

        // Find income
        key = Random();                          // Draw random number
        myincome = income.lookUp(key);           // Look up income in table
        sprintf(incomebuffer, "%ld", myincome);  // Write income to buffer
```

```
        // Hash and look up hobby
        key = Whash(2, jobbuffer, 0.3, incomebuffer, 0.5);
        hobbystring = hobby.lookUp(key);

        // Write hobby to standard output
        cout << hobbystring << endl;
    }

    return 0;
};
```

Figure 6 shows a plot of hobbies for CLERK similar to those in Fig. 5. Again a fair number of clerks have CRAFTS as a hobby, and some participate in other hobbies with a distribution the shape of which reflects that of the input frequency table. The addition of the income dependency, however, only altered the order of popular hobbies. Since we chose income independently from occupation, the income hash affects the hobby distribution in the same manner, regardless of occupation.

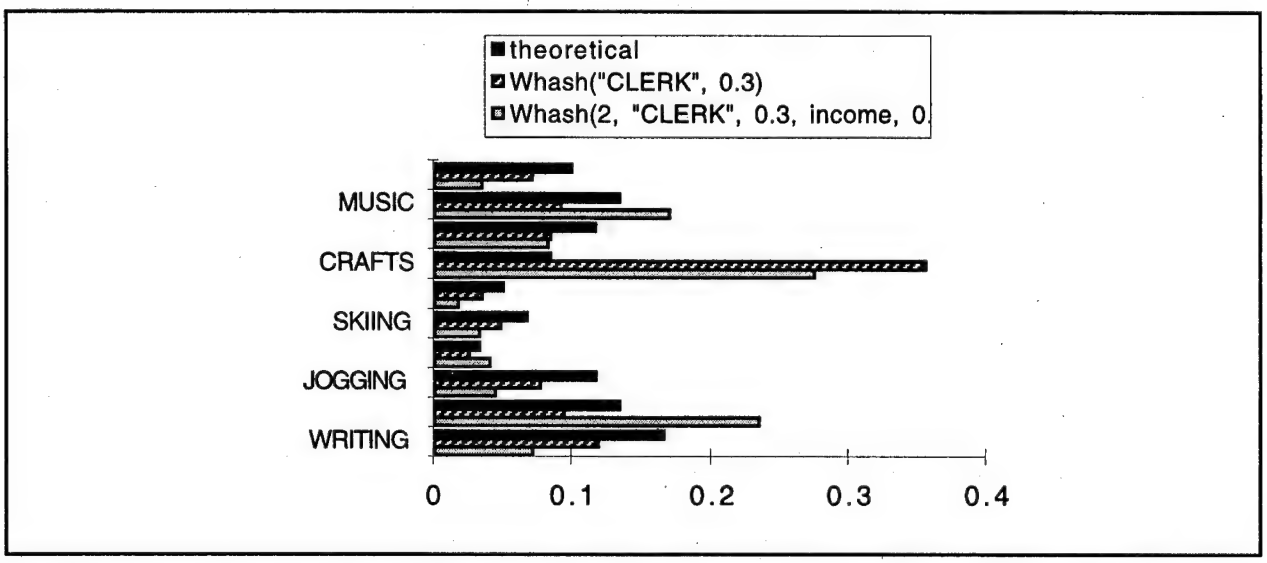

Fig. 6 — Hobbys produced by using Whash(2, "CLERK", 0.3, occupation, 0.5) and Whash("CLERK", 0.3) compared with the theoretical input hobby frequencies.

In practise, we probably wouldn't consider occupation and average income independent variables. Instead, we might say that an average income usually depends on occupation. In this case, we might have the following field relationships:

OCCUPATION * PROBABILITY OI => INCOME

(OCCUPATION * PROBABILITY OH) + (INCOME * PROBABILITY IH) => HOBBY

For instance, if the average income depends on occupation 75% of the time, then instead of employing key = Random() in our program to find the income, we would use the hash routine key = Whash(occupation, 0.75).

With these table look-up and hash methods, we have explored and implemented techniques that generate field data from frequency tables and have incorporated the emulation of field dependencies. Along the way, I also employed the process of "randomization" with respect to numeric data. Next, I will introduce the concept of randomization of character string or text data through key entry errors.

Key Entry Errors

We can always randomize text data via synonyms. An occupation data field containing DOCTOR could also have contained DR, MD, MEDICAL DOCTOR, INTERNIST, NEUROLOGIST, PATHOLOGIST and so on. We cannot reduce the simulation of synonym differences with a mere function. We can only enrich text data in this manner by compiling a complete set of data tables to satisfy the particular needs of our problem. There is, however, another form of text randomization that can be simulated with a few simple functions. This is the key entry error, where perfectly good database analysis programs may be caught unaware by the idiosyncrasies of the data entry process. As in real life DOCTOR might appear as DOCTR, DOCT OR, DOCTRO, DOCTORD, DOCTOE or OCTOR.

Let us now focus our attention on the simulation of typical errors created by entering data from a standard keyboard. I identify four basic forms of key entry errors.

1. Substitution
2. Addition
3. Deletion
4. Transposition.

Substitution occurs when one key is replaced with another. This is usually a neighboring key. For instance, the name JAMES could be typed as HAMES, substituting the key H for J.

Our algorithm for substitution is:

1. *Randomly choose a letter from the text string.*
2. *Choose the new letter.*
3. *Replace the old letter with the new letter.*

When an extra key is pressed we have addition. This is usually the same key, but can also be a neighboring key. As an example, JAMES could be entered as JJAMES.

For addition, we use the following algorithm:

1. *Randomly choose a letter from the text string.*
2. *Choose another letter.*
3. *Place the new letter after the selected letter in the text string.*

In deletion, a key is simply omitted. JAMES could look like AMES or JAMS. The former deletion instance, AMES, would challenge Soundex algorithms as the first letter must usually be the same for a match.

The deletion algorithm is simple:

1. *Randomly choose a letter from the text string.*
2. *Rewrite the string without the chosen letter.*

Transposition is the case when the position of two adjacent keys are interchanged. We might have JAMSE instead of JAMES.

Our transposition algorithm goes as follows:

1. *Randomly choose a letter in the text string.*
2. *If the letter is in the last position, do nothing.*
3. *Switch the position of the chosen letter with the letter to its immediate right.*

How do we choose new keys in the case of addition and substitution? We do this by creating a distribution of likely keys from a standard QWERTY key board. To go about this, we first map the ASCII character set into our QWERTY position values that are set up on a grid, Fig. 7.

` 0	**1** 1	**2** 2	**3** 3	**4** 4	**5** 5	**6** 6	7 7	**8** 8	**9** 9	**0** 10	- 11	= 12
	tab 21	**Q** 22	**W** 23	**E** 24	**R** 25	**T** 26	**Y** 27	**U** 28	**I** 29	**O** 30	**P** 31	[32
			A 43	**S** 44	**D** 45	**F** 46	**G** 47	**H** 48	**J** 49	**K** 50	**L** 51	; 52
				Z 64	**X** 65	**C** 66	**V** 67	**B** 68	**N** 69	**M** 70	, 71	

Fig. 7 — The QWERTY keyboard and corresponding position values.

In each square in the grid (see Fig. 7) the key value is on the left and its QWERTY position value is on the right. Using this numbering scheme, we can easily choose keys to the left, right, top, top left, top right, bottom, bottom left and bottom right. Notice that we have chosen to include some punctuation marks plus the tab key. These key choices allow us to test query functions with possible errors that are not strictly alphabetic. In our implementation, we have chosen the tab characters as our field delimiter, so the introduction of unwanted tabs will simulate shifted fields as well. If these keys as sources of possible errors are not desired, their elimination is a simple matter. To eliminate them, we will reduce the QWERTY grid.

Here is a function that maps characters to the QWERTY grid values:

```
int char2qwerty(char c){
 switch(c){
    case '1':
    case '!': return 1;
    case '2':
    case '@': return 2;
    case '3':
    case '#': return 3;
    case '4':
    case '$': return 4;
    case '5':
    case '%': return 5;
```

```
case '6':
case '^': return 6;
case '7':
case '&': return 7;
case '8':
case '*': return 8;
case '9':
case '(': return 9;
case '0':
case ')': return 10;
case '-':
case '_': return 11;
case '=':
case '+': return 12;
case '\t': return 21;
case 'a':
case 'A': return 43;
case 'b':
case 'B': return 68;
case 'c':
case 'C': return 66;
case 'd':
case 'D': return 45;
case 'e':
case 'E': return 24;
case 'f':
case 'F': return 46;
case 'g':
case 'G': return 47;
case 'h':
case 'H': return 48;
case 'i':
case 'I': return 29;
case 'j':
case 'J': return 49;
case 'k':
case 'K': return 50;
case 'l':
case 'L': return 51;
case 'm':
case 'M': return 70;
case 'n':
case 'N': return 69;
case 'o':
case 'O': return 30;
case 'p':
case 'P': return 31;
case 'q':
case 'Q': return 22;
```

```
    case 'r':
    case 'R': return 25;
    case 's':
    case 'S': return 44;
    case 't':
    case 'T': return 26;
    case 'u':
    case 'U': return 28;
    case 'v':
    case 'V': return 67;
    case 'w':
    case 'W': return 23;
    case 'x':
    case 'X': return 65;
    case 'y':
    case 'Y': return 27;
    case 'z':
    case 'Z': return 64;
    case '[':
    case '{': return 32;
    case ';':
    case ':': return 52;
    case ',':
    case '<': return 71;
    default : break;
  }
    return -99;
};
```

The char2qwerty() function consists of a switch statement that maps both upper and lower case letters to their QWERTY grid or postion values. To map them back to characters, we have two functions: qwerty2upper() and qwerty2lower(). The first function returns upper case characters, while the second function produces lower case characters.

```
char qwerty2upper(int qwert){
  switch (qwert){
    case 1: return '!';
    case 2: return '@';
    case 3: return '#';
    case 4: return '$';
    case 5: return '%';
    case 6: return '^';
    case 7: return '&';
    case 8: return '*';
    case 9: return '(';
    case 10: return ')';
    case 11: return '_';
    case 12: return '+';
    case 21: return '\t';
    case 22: return 'Q';
```

```
    case 23:  return 'W';
    case 24:  return 'E';
    case 25:  return 'R';
    case 26:  return 'T';
    case 27:  return 'Y';
    case 28:  return 'U';
    case 29:  return 'I';
    case 30:  return 'O';
    case 31:  return 'P';
    case 32:  return '{';
    case 43:  return 'A';
    case 44:  return 'S';
    case 45:  return 'D';
    case 46:  return 'F';
    case 47:  return 'G';
    case 48:  return 'H';
    case 49:  return 'J';
    case 50:  return 'K';
    case 51:  return 'L';
    case 52:  return ':';
    case 64:  return 'Z';
    case 65:  return 'X';
    case 66:  return 'C';
    case 67:  return 'V';
    case 68:  return 'B';
    case 69:  return 'N';
    case 70:  return 'M';
    case 71:  return '<';
    default:  break;
    }
 return ' ';
};

char qwerty2lower(int qwert){
 switch (qwert){
     case 1:  return '1';
     case 2:  return '2';
     case 3:  return '3';
     case 4:  return '4';
     case 5:  return '5';
     case 6:  return '6';
     case 7:  return '7';
     case 8:  return '8';
     case 9:  return '9';
     case 10:  return '0';
     case 11:  return '-';
     case 12:  return '=';
     case 21:  return '\t';
     case 22:  return 'q';
     case 23:  return 'w';
```

```
    case 24: return 'e';
    case 25: return 'r';
    case 26: return 't';
    case 27: return 'y';
    case 28: return 'u';
    case 29: return 'i';
    case 30: return 'o';
    case 31: return 'p';
    case 32: return '[';
    case 43: return 'a';
    case 44: return 's';
    case 45: return 'd';
    case 46: return 'f';
    case 47: return 'g';
    case 48: return 'h';
    case 49: return 'j';
    case 50: return 'k';
    case 51: return 'l';
    case 52: return ';';
    case 64: return 'z';
    case 65: return 'x';
    case 66: return 'c';
    case 67: return 'v';
    case 68: return 'b';
    case 69: return 'n';
    case 70: return 'm';
    case 71: return ',';
    default: break;
    }
 return ' ';
};
```

To find a key to the left or to the right, we merely subtract or add 1 to a key's QWERTY position value respectively. These keys have the greatest probability of being selected in the case of substitution. If there is no key to the left or to the right, then we use a space as the default.

Next, most probable key error is choosing a key above or below the desired key. We indicate the top key by subtracting 20 and the bottom key by adding 20 to a key's QWERTY position value. If we choose a key below the bottom row of keys, we select the spacebar. Again, if there is no top or bottom key, we default to the spacebar.

Finally, we have the lowest probability of stringing the top left, top right, bottom left, and bottom right keys. To choose the top left and bottom left keys, we subtract 21 and add 19 respectively. To choose the top right key, we subtract 19 and to choose the bottom right key we add 21.

Since left = -1, right = +1, top = -20, bottom = +20, therefore top left = top + left = -20 -1 = -21, bottom left = bottom + left = +20 -1 = +19, top right = top + right = -20 + 1 = -19, and bottom right = bottom + right = +20 +1 = +21.

Equipped with our QWERTY grid or position methods, here are some C language implementations of our four key entry errors: substitution, addition, deletion, and transposition. All of these functions take a character strings as arguments and return character pointers.

```
char* replace(char* name){
    int i, j;
    int length;
    int letter;
    int value;

    char newname[MAX_NAME];
    char temp[MAX_NAME];
    char add;

    float choice;

    strcpy(temp,name);

    // Select a letter out of the string
    length = strlen(temp);
    letter = (int)((length - 1) * Random());
    value = char2qwerty(temp[letter]);

    // choose letter to replace the old letter
    choice = Random();

    // if random number <= 0.5, draw again
    if (choice <= 0.5){
        choice = Random();
        // if random number <= 0.5, go left
        if (choice <= 0.5)
            value--;
        // ... else, go right
        else
            value++;
    }
            ;
    // choose another random number
    choice = Random();

    // if random number <= 0.5, draw again
    if (choice <= 0.5){
        choice = Random();
        // if random number <= 0.5, go up
        if (choice <= 0.5)
            value -= 20;
        // ... else, go down
            value += 20;
    }
```

```
    // Choose upper or lower case
    choice = Random();
    if (choice <= 0.5)
        add = qwerty2upper(value);
    else
        add = qwerty2lower(value);

    for (i = 0; i < letter; i++)
        newname[i] = temp[i];

    newname[i++] = add;

    for (j = letter + 1; j < length; j++, i++)
        newname[i] = temp[j];

    newname[i] = '\0';

    return (newname);
};
```

The replace() function performs substitution of a letter. It selects a letter from the string and then draws a random number. If the random number is less than one, the function creates a new random number. If this number is less than 0.5, we select the left key and if not, we select the right key. The function repeats this process to choose a top or bottom key. It also generates a random number to select an upper or lower case letter, before substituting the new letter in the string.

```
    char* addone(char* name){
    int i, j;
    int length;
    int letter;
    int value;

    char newname[MAX_NAME];
    char temp[MAX_NAME];
    char add;

    float choice;

    strcpy(temp,name);

    length = strlen(temp);

    // Select letter out of string
    letter = (int)((length - 1) * Random());
    value = char2qwerty(temp[letter]);

    // choose type of letter addition from distribution
    choice = Random();
```

```
    // if random number <= 0.5, draw again.
    if (choice <= 0.5){
        choice = Random();
        // if random number <= 0.5, go left
        if (choice <= 0.5){
            value--;
        }
        // ... else go right
        else {
            value++;
        }
    }

    // Choose another random number
    choice = Random();

    // if random number <= 0.5, draw again.
    if (choice <= 0.5){
        choice = Random();
        // if random number <= 0.5, go up
        if (choice <= 0.5){
            value-= 20;
        }
        // ... else go down
        else {
            value += 20;
        }
    }

    // choose upper or lower case
    choice = Random();
    if (choice <= 0.5)
        add = qwerty2upper(value);
    else
        add = qwerty2lower(value);


    for (i = 0; i < letter; i++)
        newname[i] = temp[i];

    newname[i++] = add;

    for (j = letter; j < length; j++, i++)
        newname[i] = temp[j];

    newname[i] = '\0';

    return (newname);
};
```

The addone() function implements the addition key entry error by adding one letter to the string. This function's key selection methods are very similar to those described for the replace() function above.

```
char* dropone(char* name){
    int i, j;
    int length;
    int letter;

    char newname[MAX_NAME];
    char temp[MAX_NAME];

    strcpy(temp,name);

    length = strlen(temp);

    // choose a random letter from the string
    letter = (int)((length - 1) * Random());

    // copy all letters preceding the chosen letter
    for (i = 0; i < letter; i++)
        newname[i] = temp[i];

    // copy over the letters following the chosen letter
    for (j = letter + 1; j < length; j++, i++)
        newname[i] = temp[j];

    newname[i] = '\0';

    return (newname);
};
```

The dropone() function returns a character string with a single character deleted. It chooses the letter to drop by using the random number generator and then copies all of the letters from the argument string to the return string with the exception of the letter that was selected for deletion.

```
char* transpose(char* name){
    int i;
    int length;
    int range;
    int letter;

    char newname[MAX_NAME];
    char temp[MAX_NAME];

    strcpy(temp,name);

    // choose a random position in the string
    length = strlen(temp);
    range = length - 2;
    letter = (int)(range * Random());
```

```
    // copy over all the letters up to the random position
    for (i = 0; i < letter; i++)
        newname[i] = temp[i];

    // copy the next letter after this
    newname[i++] = temp[letter + 1];

    // now copy the previous letter
    newname[i++] = temp[letter];

    // copy the remaining letters in the string
    for (i = letter + 2; i < length; i++)
        newname[i] = temp[i];

    newname[i] = '\0';

    return (newname);
};
```

Finally, the transpose function effects a single transposition and operates in a fashion similar to the dropone() function.

I combine all of the key entry functions into a single function called misspell(). The misspell() function generates a random integer between 0 and 3 and then uses a simple switch statement to select the error method.

```
char* misspell(char* name){
    int choice;

    choice = (int)Rrange((long)0, (long)4);
    switch (choice){
        case 0 : return addone(name);
        case 1 : return dropone(name);
        case 2 : return replace(name);
        case 3 : return transpose(name);
        default : break;
    }
    return name;
};
```

As an example, here is a sample program that prints "COMPUTER PROGRAMMER" to standard output 100 times. It has an error or typographical rate of 0.2. Before the program prints the string, it draws a random number between 0 and 1.0. If the number is less than or equal to the error rate of 0.2, it calls the misspell() function and prints the result. Otherwise, it just prints the string with no changes.

```
int main() {
    unsigned int myseed = 32;                    // seed for random number generator


    int i;
    int trials = 100;                            // number of trials

    double key;                                  // random number between 0 and 1
    double typos = 0.2;                          // error rate

    char mystring[SIZE];                         // character buffer

    char* job = "COMPUTER PROGRAMMER";

    // Seed the generator
    Srandom(myseed);

    for (i = 0; i < trials; I++){
        // Draw a random number between 0 and 1.0
        key = Random();

        // If the random number is less than the error rate
        if (key <= typos){
          strcpy(mystring, misspell(job));       // misspell the string & copy to buffer
          cout << mystring << endl;              // print buffer to standard output
        }
        // ... else just print the job string
        else
            cout << job << endl;
    }

    return 0;
};
```

This is a list of the unique strings produced by this program:

```
CIMPUTER PROGRAMMER
CO PUTER PROGRAMMER
COMPTER PROGRAMMER
COMPUFTER PROGRAMMER
COMPUTE RPROGRAMMER
COMPUTER  PROGRAMMER
COMPUTER PORGRAMMER
COMPUTER PR-OGRAMMER
COMPUTER PROfRAMMER
COMPUTER PROG6RAMMER
COMPUTER PROGAMMER
COMPUTER PROGR MMER
COMPUTER PROGRA MMER
```

COMPUTER PROGRA<MMER
COMPUTER PROGRAMEMR
COMPUTER PROGRAMER
COMPUTeR PROGRAMMER
COMPUTER PROGRSAMMER
COMPUTER RPOGRAMMER
COMPUTRE PROGRAMMER
COMPUTWR PROGRAMMER
COMUTER PROGRAMMER

The addition of misspelling errors increases the total number of strings for occupations in our data. Serving as outliers, these misspelled strings will have very low frequencies with respect to the strings from which we derived them. This allows us to simulate the long tails so often typical of text string database fields.

This typographical error generation method, of course, does not allow us to choose keys other than nearest neighbors or the space key. For instance, we do not allow for the selection of keys by the opposing hand. As an example, based on touch typing methods, the second finger of the right hand rests over the K key. There is some likely probability of wires getting crossed causing the typist to switch sides and use the second finger of the left hand position over the D key. Realistically, D should have some probability of replacing K. We also do not simulate any bias for the direction of key errors for the left and right hand.

CONCLUSIONS

The synthetic database methods that I have examined and implemented in this report allow us to manufacture text string and alphanumeric data fields with control over their interdependencies and probability distributions. This provides us with more useful data for the evaluation of the relational database designs and the tools we develop to analyze them.

I based most of these methods on a random number generator that produces floating point values from 0 to 1.0 and employed the ANSI C rand() function in the supporting source code. This function, although ANSI standard, is not necessarily portable as it depends on the maximum random number, RAND_MAX. This number is machine and compiler dependent, and may produce different floating point values from platform to platform. To separate the random number generator from the synthetic database functions, I provided Random() and Srandom() as an interface so that other random number generators may be "plugged in" and "played" without having to alter the rest of the source code.

The Randomize() and Rrange() functions let us generate alphanumeric deviates from an average value and from a range of values respectively. We found that the Randomize() function ran approximately twice as fast as the *Numerical Recipes* gasdev()-based Gaussian() function [2]. When we do not require deviates from a particular continuous distribution, the Randomize() and Rrange() functions offer quite a bit of speed up to our synthetic data generator. This is especially useful, if we need to create very large synthetic data tables.

In addition, the distribution or look-up table techniques allowed us to create more meaningful data fields for both text string and alphanumeric field types, rather than populate our synthetic database with uniformly drawn entries. I identified two table types: a standard table returning exact values and an interval table resulting in random data from a range of values. The look-up methods

for both table classes used a binary search algorithm rather than a sequential search. This enabled us to incorporate sizeable input frequency data tables without a dramatic slowdown at runtime.

We were also able to reflect fairly complicated dependencies between data fields by using hash functions to create our look-up keys. These hash methods produce many-to-one mappings between tables. In some cases, when we use the Dhash() function, and there is no associated probability, we may not completely cover the full range of items from a given table. In this event, we may need to use the value returned from the hash function to reseed our random number generator, spreading out the range of the look-up keys we produce. This is not necessary, of course, if we only use the Whash() function so that there is always some likelihood that we may find any item in a given distribution table.

To randomize text data, I introduced a simple key entry error algorithm that mimics typical spelling variations from a generic keyboard. This gave us the capability of randomizing text data, producing its classic long tails without having to explicitly incorporate this into our input tables. Without these key entry error functions, we would have to amass huge input files in order to reproduce the contribution of typographical errors to the database.

Together, all of these synthetic database techniques work to simulate more realistic looking data. We can introduce data dependencies between the fields of a record using hash methods and automatically generate key entry errors. To apply these methods, we need only identify the relations between the fields of a record and assemble the data frequency input tables to describe them.

ACKNOWLEDGEMENTS

I would like to thank Dr. Joseph B. Collins (NRL) for his help in probability theory, the development of the distribution table format, search algorithms, and in the use of hash functions to simulate data dependencies. I would also like to thank Jerry L. Gorline (NRL) for his assistance in the application of computational methods for probability and statistics.

REFERENCES

1. Donald E. Knuth, *The Art of Computer Programming*, **3**, Sorting and Searching, pp. 391.

2. W. H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing* (Cambridge Univ. Pr., New York, NY, 1988) pp. 217

3. Bjarne Stroustrup, *The C++ Programming Language*, 2nd ed. pp 66.

4. T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduction to Algorithms*, (The MIT Press, 1994), pp. 12-15.

5. Robert Sedgewick, *Algorithms in C++* (Princeton Univ., Addison-Welsey Publishing Co., Inc., 1992), pp. 232 - 234.

Appendix A
RANDOM NUMBER FUNCTIONS

This appendix contains the C++ source code for random number functions. It includes the random number generator and its seeding routines, randomize, random range, and hash functions. We have divided the code into two files: the header file, random.h and the source file, random.c.

random.h

```
#ifndef RANDOM_H
#define RANDOM_H

#include <iostream.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <math.h>

extern const int buffersize;                    // hash buffer size

extern unsigned Srandom(unsigned myseed);       // seed random number generator

extern double Random();                         // generate random number between
                                                // 0 and 1.

extern double Random(unsigned myseed);          // return random number between
                                                // 0 and 1 using the argument seed.

extern double Rrange(double d1, double d2);     // return random number between
                                                // d1 and d2.

extern long Rrange(long d1, long d2);           // return random number between
                                                // d1 and d2.

extern double Randomize(double d);              // randomize argument value

extern long Randomize(long d);                  // randomize argument value

extern unsigned Hash(char* mystring);           // hash function

// variable argument hash function uses the form
// double Hash(int strc, char* string1, char* string2, ...);
extern unsigned Hash(int strc, ...);

extern double Dhash(char* mystring);            // hash function returns double
```

```
// variable argument hash function returns double and uses the form
// double Dhash(int strc, char* string1, char* string2, ...);
extern double Dhash(int strc,  ...);

extern double Whash(char* mystring, double w);       // weighted hash returns double

// variable argument weighted hash function uses the form
// double Whash(int strc, char* string1, double weight1, char* string2, double
//  weight2, ...);
extern double Whash(int strc, ...);

#endif RANDOM_H
```

random.c

```
#include "random.h"

const int buffersize = 512;

/
// Srandom function
/
unsigned Srandom(unsigned myseed){
        srand(myseed);
        return myseed;
};

/
// Random function
/
double Random(){
        return (double)rand()/(double)RAND_MAX;
};

/
// Random function using argument seed
/
double Random(unsigned myseed){
        srand(myseed);
        return (double)rand()/(double)RAND_MAX;
};

/
// double Rrange function
/
double Rrange(double d1, double d2){
        double min;
        double delta;
        double rn;
```

```
        min = (d1 < d2) ? d1 : d2;

        delta = fabs(d1 - d2);

        rn = Random();

        rn *= delta;
        rn += min;

        return rn;
};

//
// long Rrange function
//
long Rrange(long d1, long d2){
        return ((long)Rrange((double)d1,(double)d2));
};

//
// double Randomize function
//
double Randomize(double d){
        double value;
        value = Random() - 0.5;
        value *= (2.0 * sqrt(d));
        value += d;
        return value;
};

//
// long Randomize function
//
long Randomize(long d){
        return ((long)Randomize((double)d));
};

//
// Hash function
//
unsigned Hash(char* mystring){
        unsigned j;
        unsigned keysize;
        unsigned h;

        keysize = (unsigned)strlen(mystring);
        h = (unsigned)mystring[0];

        for (j = 1; j < keysize; j++)
                h = (h*37) + (unsigned)mystring[j];
```

```
        h %= RAND_MAX;

        return h;
};

/
// variable argument Hash function
/
unsigned Hash(int strc, ...){
        int i;
        int strings;
        va_list ap;
        char** array;
        char catbuffer[buffersize];

        // set the number of strings
        strings = strc;
        if (strings <= 0)
                return 0;

        // allocate the string array
        array = new char*[strings];

        // start the variable argument list from strc
        va_start(ap, strc);

        // read the argument strings into the string array
        for (i = 0; i < strings; i++){
          array[i] = va_arg(ap,char*);
        }

        // concatenate the strings into the buffer
        strcpy(catbuffer,array[0]);
        for (i = 1; i < strings; i++)
                strcat(catbuffer,array[i]);

        // end the variable argument list
        va_end(ap);

        // delete the string array
        delete[] array;

        return Hash(catbuffer);
};

/
// Dhash function
/
```

```
double Dhash(char* mystring){
        unsigned h;
        double r;

        h = Hash(mystring);

        r = (double)h/(double)RAND_MAX;

        return r;
};

/
// Dhash function
/
double Dhash(int strc, ...){
        int i;
        int strings;
        va_list ap;
        char** array;
        char catbuffer[buffersize];

        // set the number of strings
        strings = strc;
        if (strings <= 0)
                return 0;

        // allocate the string array
        array = new char*[strings];

        // start the variable argument list from strc
        va_start(ap,strc);

        // read the argument strings into the string array
        for (i = 0; i < strings; i++){
                array[i] = va_arg(ap,char*);
        }

        // concatenate the strings into the buffer
        strcpy(catbuffer,array[0]);
        for (i = 1; i < strings; i++)
                strcat(catbuffer,array[i]);

        // end the variable argument list
        va_end(ap);

        // delete the string array
        delete[] array;

        return Dhash(catbuffer);
};
```

```
/
// Whash function
/
double Whash(char* mystring, double w){
        unsigned h;
        double r, rn;

        if (w <= 0.0 || w > 1.0){
                cerr << "weight must be in range (0,1]" << endl;
                exit(1);
        }

        // generate a random number
        rn = Random();

        // if random number is less than equal to the weight, then hash the string
        if (rn <= w)
                r = Dhash(mystring);

        // if not, return a random number
        else
                r = Random();

        return r;
};

/
// Whash function
/
double Whash(int strc, ...){
        int i;
        int strings;
        va_list ap;
        char** array;
        char catbuffer[buffersize];
        double rn;
        double* weight;

        // set the number of strings
        strings = strc;
        if (strings <= 0)
                return 0;

        // allocate the string and weight arrays
        array = new char*[strings];
        weight = new double[strings];

        // start the variable argument list from strc
        va_start(ap, strc);
```

```
        // read the strings and weights into their respective arrays
        for (i = 0; i < strings; i++){
                array[i] = va_arg(ap,char*);
                weight[i] = va_arg(ap,double);
        }

        // end the variable argument list
        va_end(ap);

        // Test the weights.  They must be in the range (0,1].
        for (i = 0; i < strings; i++){
                if (weight[i] <= 0.0 || weight[i] > 1.0){
                  cerr << "weight " << i << " must be in range (0,1]" << endl;
                  exit(1);
                }
        }

        // set the start of the catbuffer with the NULL character, '\0'
        catbuffer[0] = '\0';

        // for each of the strings and weights
        for (i = 0; i < strings; i++){
                // choose a random number
                rn = Random();

                // if random number is less than or equal to the weight, add
                // the string to the catbuffer.
                if (rn <= weight[i])
                        strcat(catbuffer, array[i]);
        }

        // delete the string and weight arrays
        delete[] array;
        delete[] weight;

        // if there are no strings in the buffer, then return a random number
        if (catbuffer[0] == '\0')
                return Random();

        return Dhash(catbuffer);
};
```

Appendix B
DISTRIBUTION TABLES

The complete C++ source code for distribution tables is shown here. First, we define the Stats structure for the frequency and distribution values, and then we use this structure as part of the Table and IntervalTable classes. Both the Table and IntervalTable use C++ templates, so that they can be instantiated using any data type or class. However, there is one caveat on the use of the IntervalTable class. To instantiate an IntervalTable, the class must overload the Rrange() function.

There are two files for this code: dist.h, the header file and dist.c, the source file. Since we have templated the Table classes, the function definitions appear in the header file as some compilers require it there. If your compiler needs them in the source file, place them to the dist.c instead.

As an extra note, the Table classes overload the iostream operators. They assume tab delimiters. Furthermore, the input stream operator expects the number of table elements as the first value.

dist.h

```
#ifndef DIST_H
#define DIST_H

#include "random.h"

struct Stats {
        double frequency;
        double distribution;
Stats(double f = 0.0, double d = 0.0) : frequency(f),          // constructor
        distribution(d) {};
~Stats(){};                                                    // destructor
friend ostream& operator<<(ostream& os, Stats& s);             // ostream operator
friend istream& operator>>(istream& is, Stats& s);             // istream operator
};

template <class T>
class Table {
public:
Table(int e = 0);                       // constructor
Table(char* filename);                  // file constructor
~Table();                               // destructor
int getElements() { return elements; }  // return the number of table elements
Table& normalize();                     // normalize table statistics
Table& clear();                         // set frequency & distribution to zero
const T& lookUp(double k) const;        // look up table item
int histogram(T& myitem);               // look up item in table and if found,
                                        // increment the frequency by 1.
```

```
const T& getItem(int i) const;                    // get the ith item from the table
friend ostream& operator<<(ostream& os, Table& t);  // ostream operator
friend istream& operator>>(istream& is, Table& t);  // istream operator
protected:
        int elements;                             // number of table elements or items
        T* item;                                  // pointer to table elements
        Stats* stats;                             // pointer to corresponding statistics
Table& findDistribution();                        // calculate distribution
};

template <class T>
class IntervalTable : public Table<T> {
public:
IntervalTable(int e = 0) : Table<T>(e) {};        // constructor
IntervalTable(char* filename) : Table<T>(filename) {};  // file constructor
T lookUpInterval(double k) const;                 // look up random table value
int histogram(T& myitem);                         // look for interval for item in
                                                  // table and if found, increment
                                                  // the frequency by 1.

};

/
// Table constructor
/
template <class T>
Table<T>::Table(int e){
        elements = e;
        if (elements < 0){
                cerr << "table elements < 0" << endl;
                exit(1);
        }
        else if (elements > 0){
                item = new T[elements];
                stats = new Stats[elements];
        }
        else {
                item = 0;
                stats = 0;
        }
};

/
// Table file constructor
/
template <class T>
Table<T>::Table(char* filename){
        ifstream fin;

        fin.open(filename);
```

```
        if (!fin){
          cerr << "Table constructor:  cannot open " << filename << endl;
          exit(1);
        }

        fin >> elements;

        if (elements <= 0){
          cerr << "Table constructor:  elements <= 0)" << endl;
          exit(1);
        }

        item = new T[elements];
        stats = new Stats[elements];

        for (int i = 0; i < elements; i++)
                fin >> item[i] >> stats[i];

        normalize();

        fin.close();
};

/
// Table destructor
/
template <class T>
Table<T>::~Table(){
        if (elements){
                delete[] item;
                delete[] stats;
                item = 0;
                stats = 0;
                elements = 0;
        }
};

/
// Table FindDistribution function
/
template <class T>
Table<T>&
Table<T>::findDistribution(){
        int i;
        double sum = 0.0;

        for (i = 0; i < elements; i++){
                sum += stats[i].frequency;
                stats[i].distribution = sum;
        }
```

```
        return *this;
};

//
// Table normalize function
//
template <class T>
Table<T>&
Table<T>::normalize(){
        int i;
        double nfactor;

        findDistribution();

        nfactor = stats[elements-1].distribution;

        if (nfactor == 0){
                cerr << "table normalize:  divide by zero" << endl;
                exit(1);
        }

        else {
                for (i = 0; i < elements; i++){
                        stats[i].frequency /= nfactor;
                        stats[i].distribution /= nfactor;
                }
        }
        return *this;
};

//
// Table Clear function
//
template <class T>
Table<T>&
Table<T>::clear(){
        for (int i = 0; i < elements; i++){
                stats[i].frequency = 0.0;
                stats[i].distribution = 0.0;
        }
        return *this;
};

//
// Table LookUp function
//
template <class T>
const T&
Table<T>::lookUp(double k) const {
        int minimumIndex = 0;
```

```
        int maximumIndex = elements - 1;
        int middleIndex = 0;

        if (key < 0.0 || key > 1.0){
          cerr << "Table::lookUp() - key out of range" << endl;
          exit(1);
        }

        // if key is 0.0, return item at minimum index
        if (k == 0.0)
          return (item[0]);

        // if key is 1.0, return item at maximum index
        if (k == 1.0)
          return (item[maximumIndex]);

        while (maximumIndex >= minimumIndex){
                middleIndex = (maximumIndex + minimumIndex)/2;
                if (k > stats[middleIndex].distribution)
                  minimumIndex = middleIndex+1;
                else if (k < stats[middleIndex].distribution)
                  maximumIndex = middleIndex-1;
                else {
                  return (item[middleIndex]);
                }
        }
        return (item[minimumIndex]);
};

/
// Table histogram function
/
template <class T>
int
Table<T>::histogram(T& myitem){
        int i;

        for (i = 0; i < elements; i++){
                if (item[i] == myitem){
                        stats[i].frequency += 1;
                        return 1;
                }
        }

        return 0;
};

/
// Table GetItem function
/
```

```
template <class T>
const T&
Table<T>::getItem(int i) const {
        if (i < 0 || i >= elements){
                cerr << "Table::GetItem() - index out of range" << endl;
                exit(1);
        }
        return item[i];
};

/
// Table histogram function
/
template <class T>
int
Table<T>::histogram(T& myitem){
        int i;

        for (i = 0; i < elements; i++){
                if (item[i] == myitem){
                        stats[i].frequency += 1;
                        return 1;
                }
        }

        return 0;
};

/
// Table GetItem function
/
template <class T>
const T&
Table<T>::getItem(int i) const {
        if (i < 0 || i >= elements){
                cerr << "Table::GetItem() - index out of range" << endl;
                exit(1);
        }
        return item[i];
};

/
// Table ostream operator
/
template <class T>
ostream& operator<<(ostream& os, Table<T>& t){
        if (t.elements){
          for (int i = 0; i < t.elements; i++)
            os << t.item[i] << "\t" << t.stats[i] << endl;
        }
```

```
        return os;
};

/
// Table istream operator
/
template <class T>
istream& operator>>(istream& is, Table<T>& t){
        int e;
        is >> e;

        if (e <= 0){
          cerr << "table elements <= 0" << endl;
          exit(1);
        }

        if (e != t.elements && t.elements > 0){
                delete[] t.item;
                delete[] t.stats;
        }

        t.elements = e;
        t.item = new T[t.elements];
        t.stats = new Stats[t.elements];

        for (int i = 0; i < t.elements; i++)
                is >> t.item[i] >> t.stats[i];

        t.normalize();

        return is;
};

/
// IntervalTable LookUp function
/
template <class T>
T
IntervalTable<T>::lookUpInterval(double k) const {
        int min, max;
        int minimumIndex = 0;
        int maximumIndex = elements - 1;
        int middleIndex = 0;
        T zero = 0;

        if (k < 0.0 || k > 1.0){
          cerr << "IntervalTable::lookUpInterval() - key out of range" << endl;
          exit(1);
        }
```

```
        // If key is 0.0, return zero
        if (k == 0.0)
          return zero;

        // If key is 1.0, return a random value between the maximum
        // index and the next lower index.
        if (k == 1.0){
          max = minimumIndex;
          min = max - 1;
          if (min < 0)
                return (Rrange(zero, item[max]));
          return (Rrange(item[min], item[max]));
        }

        while (maximumIndex >= minimumIndex){
                middleIndex = (maximumIndex + minimumIndex)/2;

                if (k > stats[middleIndex].distribution)
                  minimumIndex = middleIndex+1;
                else if (k < stats[middleIndex].distribution)
                  maximumIndex = middleIndex-1;
                else {
                  max = middleIndex;
                  min = max - 1;
                  if (min < 0)
                        return (Rrange(zero, item[max]));
                  return (Rrange(item[min], item[max]));
                }
        }
        max = minimumIndex;
        min = max - 1;
        if (min < 0)
          return (Rrange(zero,item[max]));
        return (Rrange(item[min], item[max]));
};

/
// IntervalTable histogram function
/
template <class T>
int
IntervalTable<T>::histogram(T& myitem) {
        for (int i = 0; i < elements; i++){
                if (myitem <= item[i]){
                        stats[i].frequency += 1;
                        return 1;
                }
        }
        return 0;
};
```

```
#endif DIST_H
```

dist.c

```
#include "dist.h"

//
// Stats ostream operator
//
ostream& operator<<(ostream& os, Stats& s){
        os << s.frequency << "\t" << s.distribution;
        return os;
};

//
// Stats istream operator
//
istream& operator>>(istream& is, Stats& s){
        is >> s.frequency;
        return is;
};
```

Appendix C
KEY ENTRY ERRORS

This appendix consists of the header file keyerr.h, and the source file keyerr.c implementing key entry error routines with the C^{++} programming language.

These functions reflect a rather generic keyboard and do not include extended keys. They may be altered to emulate a specific keyboard design. Further, a given database may not use certain characters. For instance, lower case keys. In this event, you may remove the lower case routines.

keyerr.h

```
#ifndef KEYERR_H
#define KEYERR_H

#include "random.h"

extern const int keyErrorTypes;
extern const int ASCII;
extern const int MIN;
extern const int MAX_NAME;

extern int char2qwerty(char c);          // ASCII to QWERTY conversion

extern char qwerty2upper(int qwert);  // QWERTY to upper case ASCII conversion
extern char qwerty2lower(int qwert);  // QWERTY to lower case ASCII conversion

extern char* addone(char* name);      // add one letter to the character string
extern char* dropone(char* name);     // remove one letter from the character string
extern char* replace(char* name);     // replace one letter in the character string
extern char* transpose(char* name);   // transpose two letters in the character string

extern char* misspell(char* name);    // misspell a string using one of the four error
                                      // functions above

#endif KEYERR_H
```

keyerr.c

```
#include "keyerr.h"

const int keyErrorTypes = 4;
const int ASCII = 25;
const int MIN = 97;
const int MAX_NAME = 128;
```

```
/
// char2qwerty function
/
int char2qwerty(char c){
  switch(c){
        case '1':
        case '!':  return 1;
        case '2':
        case '@':  return 2;
        case '3':
        case '#':  return 3;
        case '4':
        case '$':  return 4;
        case '5':
        case '%':  return 5;
        case '6':
        case '^':  return 6;
        case '7':
        case '&':  return 7;
        case '8':
        case '*':  return 8;
        case '9':
        case '(':  return 9;
        case '0':
        case ')':  return 10;
        case '-':
        case '_':  return 11;
        case '=':
        case '+':  return 12;
        case '\t': return 21;
        case 'a':
        case 'A':  return 43;
        case 'b':
        case 'B':  return 68;
        case 'c':
        case 'C':  return 66;
        case 'd':
        case 'D':  return 45;
        case 'e':
        case 'E':  return 24;
        case 'f':
        case 'F':  return 46;
        case 'g':
        case 'G':  return 47;
        case 'h':
        case 'H':  return 48;
        case 'i':
        case 'I':  return 29;
        case 'j':
        case 'J':  return 49;
```

```
        case 'k':
        case 'K':  return 50;
        case 'l':
        case 'L':  return 51;
        case 'm':
        case 'M':  return 70;
        case 'n':
        case 'N':  return 69;
        case 'o':
        case 'O':  return 30;
        case 'p':
        case 'P':  return 31;
        case 'q':
        case 'Q':  return 22;
        case 'r':
        case 'R':  return 25;
        case 's':
        case 'S':  return 44;
        case 't':
        case 'T':  return 26;
        case 'u':
        case 'U':  return 28;
        case 'v':
        case 'V':  return 67;
        case 'w':
        case 'W':  return 23;
        case 'x':
        case 'X':  return 65;
        case 'y':
        case 'Y':  return 27;
        case 'z':
        case 'Z':  return 64;
        case '[':
        case '{': return 32;
        case ';':
        case ':': return 52;
        case ',':
        case '<': return 71;
        default :  break;
  }
        return -99;
};

/
// qwerty2upper function
/
char qwerty2upper(int qwert){
  switch (qwert){
        case 1:  return '!';
        case 2:  return '@';
```

```
        case 3:  return '#';
        case 4:  return '$';
        case 5:  return '%';
        case 6:  return '^';
        case 7:  return '&';
        case 8:  return '*';
        case 9:  return '(';
        case 10:  return ')';
        case 11:  return '_';
        case 12:  return '+';
        case 21:  return '\t';
        case 22:  return 'Q';
        case 23:  return 'W';
        case 24:  return 'E';
        case 25:  return 'R';
        case 26:  return 'T';
        case 27:  return 'Y';
        case 28:  return 'U';
        case 29:  return 'I';
        case 30:  return 'O';
        case 31:  return 'P';
        case 32:  return '{';
        case 43:  return 'A';
        case 44:  return 'S';
        case 45:  return 'D';
        case 46:  return 'F';
        case 47:  return 'G';
        case 48:  return 'H';
        case 49:  return 'J';
        case 50:  return 'K';
        case 51:  return 'L';
        case 52:  return ':';
        case 64:  return 'Z';
        case 65:  return 'X';
        case 66:  return 'C';
        case 67:  return 'V';
        case 68:  return 'B';
        case 69:  return 'N';
        case 70:  return 'M';
        case 71:  return '<';
        default:  break;
        }
  return ' ';
};

/
// qwerty2lower function
/
char qwerty2lower(int qwert){
  switch (qwert){
```

```
        case 1:  return '1';
        case 2:  return '2';
        case 3:  return '3';
        case 4:  return '4';
        case 5:  return '5';
        case 6:  return '6';
        case 7:  return '7';
        case 8:  return '8';
        case 9:  return '9';
        case 10:  return '0';
        case 11:  return '-';
        case 12:  return '=';
        case 21:  return '\t';
        case 22:  return 'q';
        case 23:  return 'w';
        case 24:  return 'e';
        case 25:  return 'r';
        case 26:  return 't';
        case 27:  return 'y';
        case 28:  return 'u';
        case 29:  return 'i';
        case 30:  return 'o';
        case 31:  return 'p';
        case 32:  return '[';
        case 43:  return 'a';
        case 44:  return 's';
        case 45:  return 'd';
        case 46:  return 'f';
        case 47:  return 'g';
        case 48:  return 'h';
        case 49:  return 'j';
        case 50:  return 'k';
        case 51:  return 'l';
        case 52:  return ';';
        case 64:  return 'z';
        case 65:  return 'x';
        case 66:  return 'c';
        case 67:  return 'v';
        case 68:  return 'b';
        case 69:  return 'n';
        case 70:  return 'm';
        case 71:  return ',';
        default:  break;
        }
  return ' ';
};
```

```
/
// addone function
/
char* addone(char* name){
        int i, j;
        int length;
        int letter;
        int value;

        char newname[MAX_NAME];
        char temp[MAX_NAME];
        char add;

        float choice;

        strcpy(temp,name);

        length = strlen(temp);

        // Select letter out of string
        letter = (int)((length - 1) * Random());
        value = char2qwerty(temp[letter]);

        // choose type of letter addition from distribution
        choice = Random();

        // if random number <= 0.5, draw again.
        if (choice <= 0.5){
                choice = Random();
                // if random number <= 0.5, go left
                if (choice <= 0.5){
                        value--;
                }
                // ... else go right
                else {
                        value++;
                }
        }

        // Choose another random number
        choice = Random();

        // if random number <= 0.5, draw again.
        if (choice <= 0.5){
                choice = Random();
                // if random number <= 0.5, go up
                if (choice <= 0.5){
                        value-= 20;
                }
                // ... else go down
```

```
                else {
                        value += 20;
                }
        }

        // choose upper or lower case
        choice = Random();
        if (choice <= 0.5)
                add = qwerty2upper(value);
        else
                add = qwerty2lower(value);


        for (i = 0; i < letter; i++)
                newname[i] = temp[i];

        newname[i++] = add;

        for (j = letter; j < length; j++, i++)
                newname[i] = temp[j];

        newname[i] = '\0';

        return (newname);
};

/
// dropone function
/
char* dropone(char* name){
        int i, j;
        int length;
        int letter;

        char newname[MAX_NAME];
        char temp[MAX_NAME];

        strcpy(temp,name);

        length = strlen(temp);

        letter = (int)((length - 1) * Random());

        for (i = 0; i < letter; i++)
                newname[i] = temp[i];

        for (j = letter + 1; j < length; j++, i++)
                newname[i] = temp[j];

        newname[i] = '\0';
        return (newname);
};
```

```
/
// replace function
/
char* replace(char* name){
        int i, j;
        int length;
        int letter;
        int value;

        char newname[MAX_NAME];
        char temp[MAX_NAME];
        char add;

        float choice;

        strcpy(temp,name);

        // Select a letter out of the string
        length = strlen(temp);
        letter = (int)((length - 1) * Random());
        value = char2qwerty(temp[letter]);

        // choose letter to replace the old letter
        choice = Random();

        // if random number <= 0.5, draw again
        if (choice <= 0.5){
                choice = Random();
                // if random number <= 0.5, go left
                if (choice <= 0.5)
                        value--;
                // ... else, go right
                else
                        value++;
        }

        // choose another random number
        choice = Random();

        // if random number <= 0.5, draw again
        if (choice <= 0.5){
                choice = Random();
                // if random number <= 0.5, go up
                if (choice <= 0.5)
                        value -= 20;
                // ... else, go down
                        value += 20;
        }

        // Choose upper or lower case
```

```
        choice = Random();
        if (choice <= 0.5)
                add = qwerty2upper(value);
        else
                add = qwerty2lower(value);

        for (i = 0; i < letter; i++)
                newname[i] = temp[i];

        newname[i++] = add;

        for (j = letter + 1; j < length; j++, i++)
                newname[i] = temp[j];

        newname[i] = '\0';

        return (newname);
};

/
// transpose function
/
char* transpose(char* name){
        int i;
        int length;
        int range;
        int letter;

        char newname[MAX_NAME];
        char temp[MAX_NAME];

        strcpy(temp,name);

        length = strlen(temp);
        range = length - 2;
        letter = (int)(range * Random());

        for (i = 0; i < letter; i++)
                newname[i] = temp[i];

        newname[i++] = temp[letter + 1];
        newname[i++] = temp[letter];

        for (i = letter + 2; i < length; i++)
                newname[i] = temp[i];

        newname[i] = '\0';

        return (newname);
};
```

```
/
// misspell function
/
char* misspell(char* name){
        int choice;

        choice = (int)Rrange((long)0, (long)4);
        switch (choice){
                case 0 : return addone(name);
                case 1 : return dropone(name);
                case 2 : return replace(name);
                case 3 : return transpose(name);
                default : break;
        }
        return name;
};
```

Appendix D
A SIMPLE STRING CLASS

Most C++ libraries provide a String class to handle character strings. This elementary String class serves as a portable substitute if your compiler does not include this class. Additionally, as defined by our Table classes (see Appendix B), the iostream operators assume a tab character delimiter. You must keep the overloading of these operators in mind with respect to the delimiters required by your database. We have divided the code for the String class into two files: the header file, mystring.h, and the source file, mystring.c.

mystring.h

```
#ifndef MYSTRING_H
#define MYSTRING_H

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>

extern const int buffer;

/
// A simple string class
/
class String {
        int length;                                          // length of string
        char* field;                                         // character string
public:
String(char* s = "");                                        // default constructor
String(const String& s);                                     // copy constructor
~String();                                                   // destructor
ostream& print(ostream& os) const;                           // print string field to ostream
String& operator=(const String& s);                          // assignment operator
int operator==(const String& s) const;                       // equality operator
friend ostream& operator<<(ostream& os, const String& s);    // ostream operator
friend istream& operator>>(istream& is, String& s);          // istream operator
const char* getString() const;                               // return string field
};

#endif MYSTRING_H
```

mystring.c

```
#include "mystring.h"
```

```
const int buffer = 64;

//
// String default constructor
//
String::String(char* s){
        length = strlen(s);
        if (length > 0){
                field = new char[length + 1];
                strcpy(field,s);
        }
        else
                field = 0;
};

//
// String copy constructor
//
String::String(const String& s){
        if (length > 0 && length != s.length){
                delete[] field;
                length = s.length;
                field = new char[length + 1];
        }
        strcpy(field,s.field);
};

//
// String destructor
//
String::~String(){
        if (length > 0){
                delete[] field;
                field = 0;
                length = 0;
        }
};

//
// String print function
//
ostream& String::print(ostream& os) const {
        if (length)
                os << field;
        return os;
};

//
// String assignment operator
//
```

```
String& String::operator=(const String& s){
        if (length > 0 || length != s.length){
                delete[] field;
                length = s.length;
                field = new char[length + 1];
        }
        strcpy(field,s.field);
        return *this;
};

//
// String equalit operator
//
int String::operator==(const String& s) const {
        if (length != s.length)
                return 0;
        if (strcmp(field, s.field))
                return 0;
        return 1;
};

//
// ostream operator overloaded for String
//
ostream& operator<<(ostream& os, const String& s){
        return s.print(os);
};

//
// istream operator overloaded for String
//
istream& operator>>(istream& is, String& s){
        int len;
        char temp[buffer];
        char newline;
        newline = is.get();
        if (newline != '\n')
                is.putback(newline);
        is.getline(temp,buffer,'\t');
        len = strlen(temp);

        if (s.length > 0 || s.length != len){
                delete[] s.field;
                s.length = len;
                s.field = new char[len + 1];
        }

        strcpy(s.field,temp);
        return is;
};
```